LOVELY PROFESSIONAL UNIVERSITY

SOFTWARE ENGINEERING

Edited By

Deepak Mehta

Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Lovely Professional University
Phagwara

SYLLABUS

Software Engineering

Objectives: To enable the student to understand software development process and software engineering approaches with testing techniques. Student will practice software engineering concepts using UML.

Sr. No.	Description
1	Introduction to Software Engineering: The Evolving Role of Software, Software Myths
2	A generic view of Process: Software Engineering-A Layered Technology, A process framework, The Capability Maturity Model Integration, Process Patterns, Process Assessment.
3	Process Models : Prescriptive Models, The Waterfall model, Incremental Process Models : The Incremental model, The RAD model Evolutionary Process models: Prototyping, The Spiral model, The Concurrent Development model, A final comment on evolutionary Processes.
4	An Agile view of Process: What is Agility, Agile Process models: XP, ASD, DSDM, Scrum, Crystal, FDD, AM. Requirements Engineering: A Brigade to design & construction, Requirements Engineering tasks: Inception, Elicitation, negotiation, Specification, Validation, Requirements Management.
5	Software Engineering Practice : The Essence of practice, Core Principles, Planning practices, Modelling practices: Analysis modelling principles, Design Modelling principles Construction practice : Coding principles and concepts, Testing principles Design Engineering: Design process & Design Quality. Design Concepts: Abstraction ,Architecture, patterns, Modularity, Information hiding, Functional independence, Refinement, Refactoring, Design Classes

6	System Engineering : The System Engineering Hierarchy: System Modelling, System Simulation, System Modelling: Hatley-Pirbhai Modelling, System Modelling with UML Creating an Architectural Design: Data design: Data design at the Architectural level & component level, Architectural Design: Representing the system in Context, Defining Archetypes, Refining the Architecture into components, Describing installations of the system.
7	Testing Strategies: - Testing strategies for conventional software, test strategies for object- oriented software, validation testing, system testing. Requirements Engineering: A Bridge to Design and Construction, Requirements Engineering Tasks: Inception, Elicitation, Elaboration, Negotiation, Specification, Validation, Requirements Management.
8	Testing Tactics: Black-box testing & white box testing, flow-graph testing, equivalence partitioning, Boundary value analysis, Fault based testing. Building the Analysis Model: Requirements Analysis: Overall objective and philosophy, Analysis rules of Thumb, Domain Analysis, Analysis Modelling approaches. Data Modelling concepts: Data objects, Data attributes, Relationships, Cardinality and Modality
9	Design Engineering : Design Process and Design quality, Design concepts: Abstraction, Architecture, patterns, Modularity, Information hiding, Functional independence, Refinement, Refactoring, Design classes
10	Creating an Architectural Design : Data design: Data design at the Architectural level and Component level, Architectural Design: Representing the system in Context, Defining Archetypes, Refining the Architecture into Components, Describing installations of the system

CONTENT

Unit 1:	The Evolving Role of Software *Deepak Mehta, Lovely Professional University*	1
Unit 2:	A Generic View of Process *Deepak Mehta, Lovely Professional University*	20
Unit 3:	Process Models *Deepak Mehta, Lovely Professional University*	37
Unit 4:	Evolutionary Process Models *Deepak Mehta, Lovely Professional University*	50
Unit 5:	An Agile View of Process *Deepak Mehta, Lovely Professional University*	62
Unit 6:	Requirement Engineering *Deepak Mehta, Lovely Professional University*	83
Unit 7:	Software Engineering Practice *Mandeep Kaur, Lovely Professional University*	98
Unit 8;	Design Engineering *Mandeep Kaur, Lovely Professional University*	116
Unit 9:	System Engineering *Mandeep Kaur, Lovely Professional University*	137
Unit 10:	Creating an Architectural Design *Deepak Mehta, Lovely Professional University*	168
Unit 11:	Testing Strategies *Deepak Mehta, Lovely Professional University*	176
Unit 12:	Testing Tactics *Deepak Mehta, Lovely Professional University*	196
Unit 13:	creating an Architectural Design *Sarabjit Kumar, Lovely Professional University*	217
Unit 14:	Data Modeling *Sarabjit Kumar, Lovely Professional University*	235

Notes

Unit 1: The Evolving Role of Software

CONTENTS

Objectives
Introduction
1.1 Evolving Role of Software
1.2 Software
1.2.1 Program vs. Software
1.2.2 Software Engineering and Its Relationship with Other Disciplines
1.3 Software Myths
1.3.1 Types of Myths
1.4 Software Crisis
1.5 Software Engineering Framework
1.5.1 Software Engineering
1.5.2 Software Process
1.5.3 Software Engineering Methods
1.5.4 Key Challenges facing Software Engineering
1.5.5 ACM/IEEE Code of Ethics
1.5.6 Difference between Software Engineering and Computer Science
1.5.7 Difference between Software Engineering and System Engineering
1.5.8 State of the Practice
1.6 Software Development Issues
1.7 Summary
1.8 Keywords
1.9 Review Questions
1.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize the evolving role of software
- Describe computer software
- Demonstrate the software myths

Introduction

The complexity and nature of software have changed tremendously in the last four decades. In the 70s applications ran on a single processor, received single line inputs and produced

Notes

alphanumeric results. However, the applications today are far more complex running on client-server technology and have a user friendly GUI. They run on multiple processors with different OS and even on different geographical machines.

The software groups work hard as they can to keep abreast of the rapidly changing new technologies and cope with the developmental issues and backlogs. Even the Software Engineering Institute (SEI) advises to improve upon the developmental process. The "change" is an inevitable need of the hour. However, it often leads to conflicts between the groups of people who embrace change and those who strictly stick to the traditional ways of working.

Notes Thus, there is an urgent need to adopt software engineering concepts, practices, strategies to avoid conflicts and in order to improve the software development to deliver good quality software within budget and time.

1.1 Evolving Role of Software

The role of software has undergone drastic change in the last few decades. These improvements range through hardware, computing architecture, memory, storage capacity and a wide range of unusual input and output conditions. All these significant improvements have lead to the development of more complex and sophisticated computer-based systems. Sophistication leads to better results but can cause problems for those who build these systems.

Lone programmer has been replaced by a team of software experts. These experts focus on individual parts of technology in order to deliver a complex application. However, the experts still face the same questions as that by a lone programmer:

- Why does it take long to finish software?
- Why are the costs of development so high?
- Why aren't all the errors discovered before delivering the software to customers?
- Why is it difficult to measure the progress of developing software?

All these questions and many more have lead to the manifestation of the concern about software and the manner in which it is developed – a concern which lead to the evolution of the software engineering practices.

Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—information.

Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., Internet) and provides the means for acquiring information in all of its forms.

Notes

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

Task The same questions asked of the lone programmer are being asked when modern computer-based systems are built. Give answers to questions:

1. Why does it take so long to get software finished?
2. Why are development costs so high?
3. Why can't we find all the errors before we give the software to customers?
4. Why do we continue to have difficulty in measuring progress as software is being developed?

1.2 Software

In the first NATO conference on software engineering in 1968, Fritz Bauer defined Software engineering as "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines". Stephen Schach defined the same as "A discipline whose aim is the production of quality software, software that is delivered on time, within budget and that satisfies its requirements".

The software differs from hardware as it is more logical in nature and hence, the difference in characteristics. Let us now explore the characteristics of software in detail.

Software is developed or engineered and not manufactured

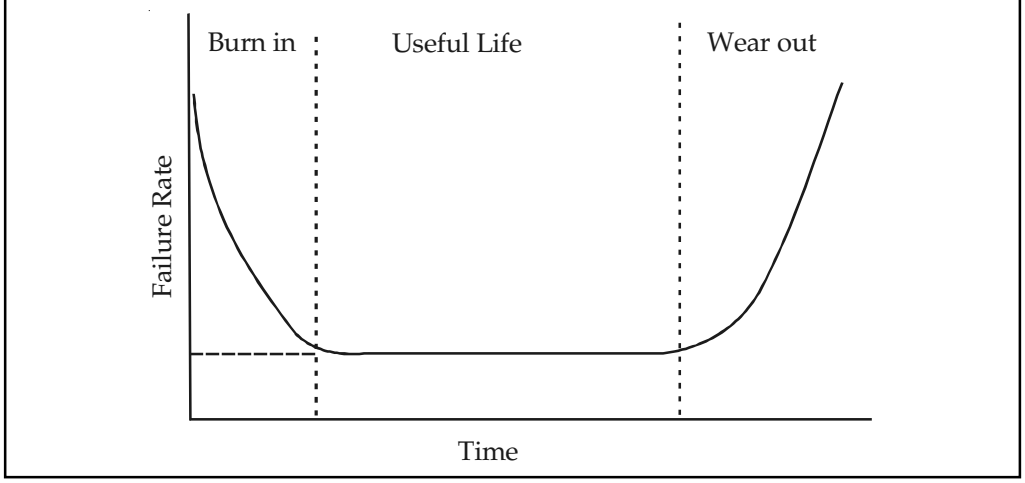
Although there exists few similarities between the hardware manufacturing and software development, the two activities differ fundamentally. Both require a good design to attain high quality. But the manufacturing phase of hardware can induce quality related problems that are either non-existent for software or can be easily rectified. Although both activities depend on people but the relationship between people and work is totally different.

Software does not wear out

Figure 1.1 shows the failure rate of hardware as a function of time. It is often called the "bathtub curve", indicating that a hardware shows high failure at its early stage (due to design and manufacturing defects); defects get resolved and the failure rate reduces to a steady-state level for some time. As time progresses, the failure rate shoots up again as the hardware wears out due to dust, temperature and other environmental factors.

Notes

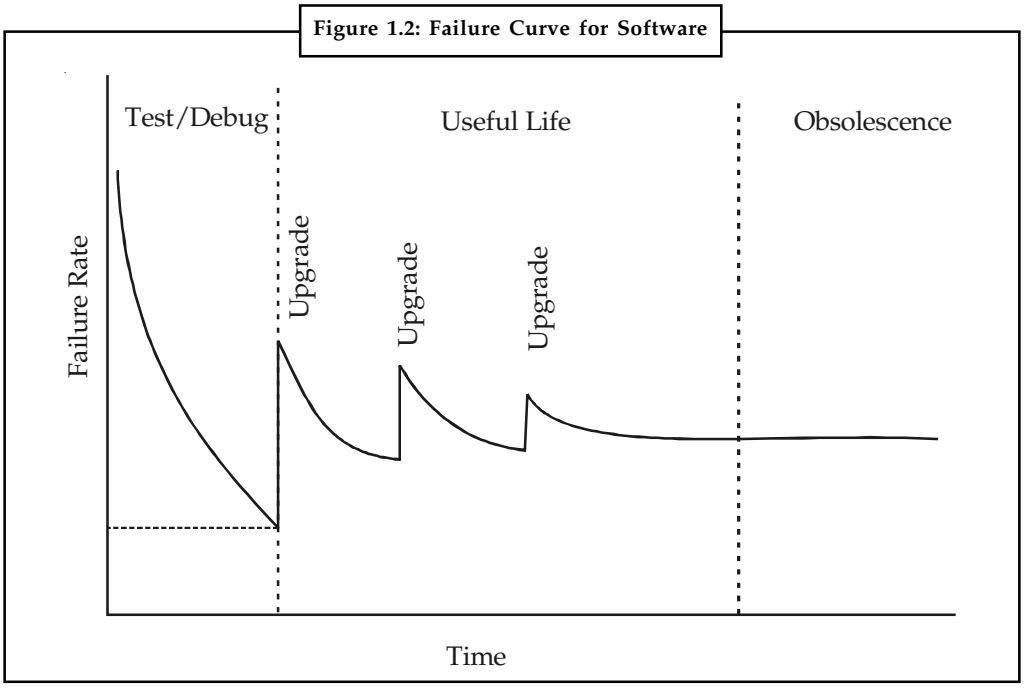
Figure 1.1: Failure Curve for Hardware



Because the software does not undergo environmental degradation, its failure curve ideally should flatten out after the initial stage of its life. The initial failure rate is high because of undiscovered defects. Once these defects are fixed, the curve flattens at a later stage. Thus, the software does not wear out but deteriorates.

The actual curve as shown in Figure 1.2 can explain the contradiction stated above. Because software undergoes changes during its lifetime as a part of the maintenance activities, new defects creep in causing the failure rate to rise (spike). While the software enters its steady state further more changes are added which induce further more defects and hence, spikes in the failure rate.

Figure 1.2: Failure Curve for Software



Notes

Did u know? **What happens to the minimum failure rates?**

The minimum failure rate rises and the software deteriorates due to induction of frequent changes.

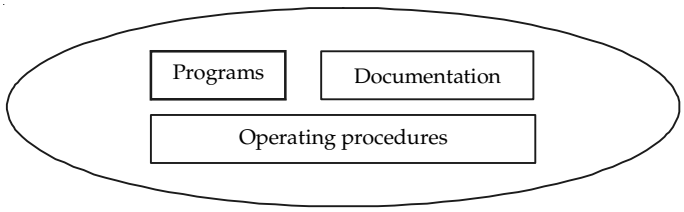
Software is Custom Built and Not Designed Component Wise

Software is designed and built so that it can be reused in different programs. Few decades ago subroutine libraries were created that re-used well defined algorithms but had a limited domain. Today this view has been extended to ensure re-usability of not only algorithms but data structures as well. Thus, the data and the processes that operate on this data were combined together to be able to use later on.

1.2.1 Program vs. Software

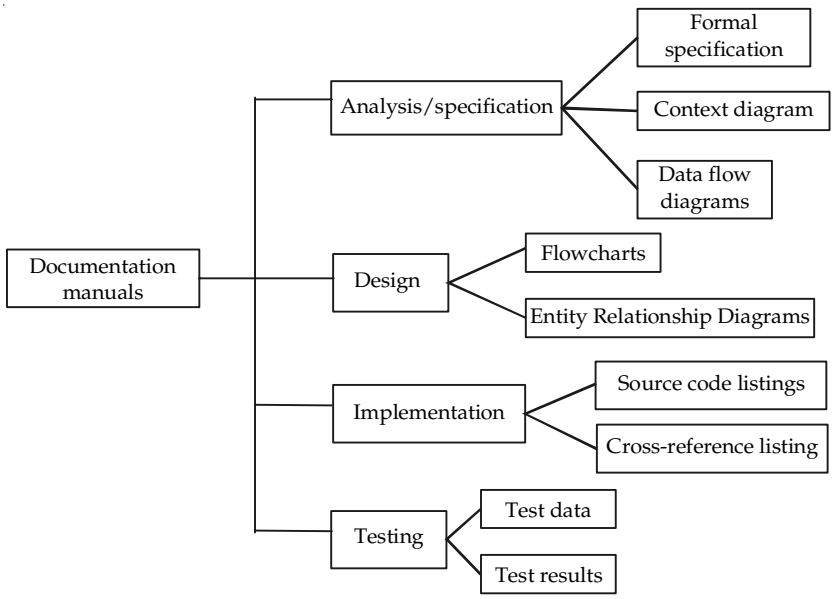
Software is more than programs. It comprises of programs, documentation to use these programs and the procedures that operate on the software systems.

Figure 1.3: Components of Software



A program is a part of software and can be called software only if documentation and operating procedures are added to it. Program includes both source code and the object code.

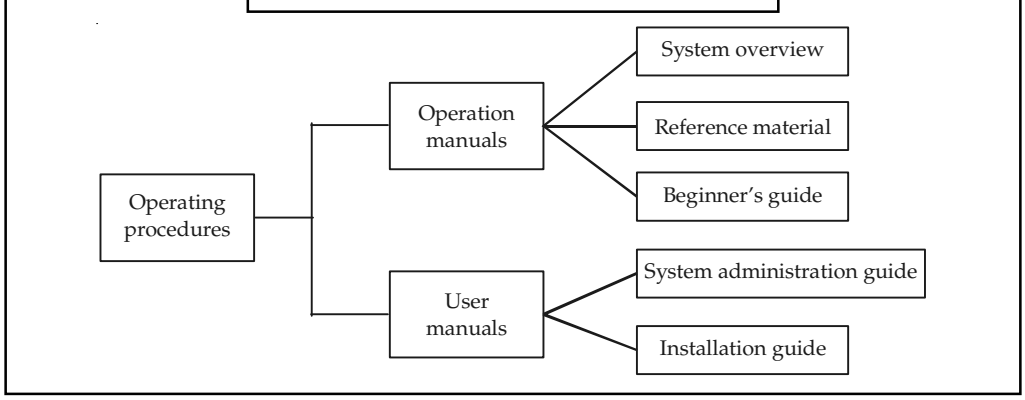
Figure 1.4: List of Documentation Manuals



Notes

Operating procedures comprise of instructions required to setup and use the software and instructions on actions to be taken during failure. List of operating procedure manuals/ documents is given in Figure 1.5.

Figure 1.5: List of Operating Procedure Manuals



1.2.2 Software Engineering and Its Relationship with Other Disciplines

Any area that involves a sequence of steps (i.e. an algorithm) to generate output can make use of software engineering. The nature of the software application depends upon two things: (1) the form of incoming and outgoing information. Some business applications generate more complex results then others and require more complex inputs. (2) The sequence and timing of the information being used. A program that takes data in a predefined order, applies algorithms to it and generates results is called a determinate program. On the other hand, the application that accepts data from multiple users at varied timings, applies logic that can be interrupted by external conditions, and generates results that vary according to time and environment are called indeterminate. Although software application cannot be categorized, following areas show the potential application of software in various breadths:

- ***Real-time software:*** It refers to the software that controls the events and when they occur. The data gatherer component of the software gathers the data and formats it into an acceptable form. The analyzer component that transforms information as required by the application. The output component that interacts with the external environment and the monitoring component that coordinates with all the components to generate timely results.
- ***Business software:*** It is the single largest area of software application. These areas use one or more large databases for business information and reframe the information in such a manner that facilitates business operations and decision making. For example, library management, data inventory, HR management, etc.
- ***System Software:*** It is the software program that interacts with the hardware components so that other application software can run on it. Operating systems like Windows, Unix, DOS, etc., are examples of the system software. Such type of software perform a wide range of operations like allocation and distribution of memory amongst various computer processes, process scheduling, resource scheduling, etc.
- ***Embedded software:*** This software resides in the commodities for the consumer and industry markets in the form of read-only memory devices. They perform limited functions e.g. keypad of an automatic washing machine or significant functions as per requirement e.g. dashboard functions in an aircraft.

Notes

- ***Personal computer software:*** These software programs have grown rapidly over the past few decades. These include the like of database management, entertainment, computer graphics, multimedia, etc.
- ***Web-based software:*** The web-pages displayed by a browser are programs that include executable instructions (e.g. CGI, Perl, Java), hypertext and audio-visual formats.
- ***Artificial intelligence software:*** AI software uses algorithms to solve problems that are not open to simple computing and analysis. For example, Pattern matching (voice or image), game playing, robotics, expert systems, etc.
- ***Engineering software:*** The software is characterized by number-crunching algorithms ranging from molecular biology, automated manufacturing to space shuttle orbit management, etc.

Self Assessment

Fill in the blanks:

1. Software transforms personal data so that the data can be more useful in a……..
2. The initial failure rate is …….. because of undiscovered defects.
3. The analyzer component that transforms …….. as required by the application.
4. The sequence and …….. of the information being used.
5. A program is a part of software and can be called software only if documentation and …… procedures are added to it.
6. Once these defects are fixed, the …….. flattens at a later stage.

1.3 Software Myths

- ***Myth 1:*** Computers are more reliable than the devices they have replaced.

  Considering the reusability of the software, it can undoubtedly be said that the software does not fail. However, certain areas which have been mechanized have now become prone to software errors as they were prone to human errors while they were manually performed. For example, Accounts in the business houses.

- ***Myth 2:*** Software is easy to change.

  Yes, changes are easy to make - but hard to make without introducing errors. With every change the entire system must be re-verified.

- ***Myth 3:*** If software development gets behind scheduled, adding more programmers will put the development back on track.

  Software development is not a mechanistic process like manufacturing. In the words of Brooks: "adding people to a late software project makes it later".

- ***Myth 4:*** Testing software removes all the errors.

  Testing ensures presence of errors and not absence. Our objective is to design test cases such that maximum number of errors is reported.

- ***Myth 5:*** Software with more features is better software.

  This is exactly the opposite of the truth. The best programs are those that do one kind of work.

Notes

- *Myth 6:* Aim has now shifted to develop working programs.

  The aim now is to deliver good quality and efficient programs rather than just delivering working programs. Programs delivered with high quality are maintainable.

The list is unending. These myths together with poor quality, increasing cost and delay in the software delivery have lead to the emergence of software engineering as a discipline.

1.3.1 Types of Myths

Software Myths

Software Myth beliefs about software and the process used to build it – can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious.

Example: For instance, myths appear to be reasonable statements of fact, they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score".

Management Myths

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, If the Belief will lessen the pressure.

Myth: 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?

Reality: The book of standards may very well exist, but is it used?

- Are software practitioners aware of its existence?
- Does it reflect modern software engineering practice?
- Is it complete? Is it adaptable?
- Is it streamlined to improve time to delivery while still maintaining a focus on Quality?

In many cases, the answer to these entire question is no.

Myth: If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept)

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks: "Adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort

Myth: If we decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software project internally, it will invariably struggle when it out sources software project.

Notes

Customer Myths

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths led to false expectations and ultimately, dissatisfaction with the developers.

Myth: A general statement of objectives is sufficient to begin writing programs we can fill in details later.

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small.

Caution However, as time passes, cost impact grows rapidly – resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Self Assessment

Fill in the blanks:

7. Software development is not a …….. process like manufacturing.
8. Project requirements continually…….. but change can be easily accommodated because software is flexible.
9. A general statement of ……. is sufficient to begin writing programs we can fill in details later.

1.4 Software Crisis

1. ***Size:*** Software is becoming larger and more complex with the growing complexity and expectations out of software. For example, the code in consumer products is doubling every couple of years.
2. ***Quality:*** Many software products have poor quality, i.e., the software produces defects after put into use due to ineffective testing techniques. For example, Software testing typically finds 25 defects per 1000 lines of code.
3. ***Cost:*** Software development is costly i.e., in terms of time taken to develop and the money involved. For example, Development of the FAA's Advance Automation System cost over $700 per line of code.
4. ***Delayed delivery:*** Serious schedule overruns are common. Very often the software takes longer than the estimated time to develop which in turn leads to cost shooting up. For example, one in four large-scale development projects is never completed.

Notes

Did u know? **What is the major cause of software crisis?**

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

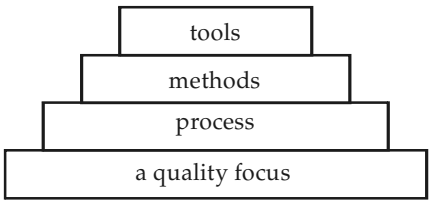
Self Assessment

Fill in the blanks:

10. Many of the software problems were caused by increasingly complex……..

1.5 Software Engineering Framework

Figure 1.6: Software Engineering Framework



Software Engineering has a three layered framework. The foundation for software engineering is the process layer. Software engineering process holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of Key Process Areas (KPA) that must be established for effective delivery of software engineering technology.

Software engineering methods provide the technical know how for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support.

Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development called Computer Aided Software Engineering (CASE) is established.

Task Analyze the advantages of CASE.

1.5.1 Software Engineering

- The economics of ALL developed nations are dependent on software.
- More and more systems are software controlled.
- Software engineering is concerned with theories, methods and tools for professional software development.

Notes

- Software engineering expenditure represents a significant fraction of GNP in all developed countries.
- Software engineering is concerned with cost-effective software development.

1.5.2 Software Process

A process is a series of steps involving activities, constraints and resources that produce an intended output of some kind.

In simpler words, when you build a product or system, it's important to go through a series of predictable steps – a road map that helps you create a timely, high quality result. The road map that you follow is called a software process.

In the last decade there has been a great deal of resources devoted to the definition, implementation, and improvement of software development processes.

- ISO 9000
- Software Process Improvement and Capability Determination (SPICE)
- SEI Processes
- Capability Maturity Model (CMM) for software
- Personal Software Process (PSP)
- Team Software Process (TSP)

A set of activities whose goal is the development or evolution of software.

Generic activities in all software processes are:

- ***Specification:*** what the system should do and its development constraints
- ***Development:*** production of the software system
- ***Validation:*** checking that the software is what the customer wants
- ***Evolution:*** changing the software in response to changing demands.

Software Process Model

A simplified representation of a software process, presented from a specific perspective.

Example: Process perspectives are:

- Workflow perspective - sequence of activities
- Data-flow perspective - information flow
- Role/action perspective – who does what

Generic Process Models

- Waterfall
- Evolutionary development
- Formal transformation
- Integration from reusable components.

Notes

1.5.3 Software Engineering Methods

Structured approaches to software development which include system models, notations, rules, design advice and process guidance.

- Model descriptions - Descriptions of graphical models, which should be produced
- Rules - Constraints applied to system models
- Recommendations - Advice on good design practice
- Process guidance - What activities to follow

CASE (Computer-Aided Software Engineering)

Software systems which are intended to provide automated support for software process activities. CASE systems are often used for method support:

Upper-CASE

Tools to support the early process activities of requirements and design.

Lower-CASE

Tools to support later activities such as programming, debugging and testing.

Attributes of Good Software

The software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.

- ***Maintainability:*** Software must evolve to meet changing needs.
- ***Dependability:*** Software must be trustworthy.
- ***Efficiency:*** Software should not make wasteful use of system resources.
- ***Usability:*** Software must be usable by the users for which it was designed.

1.5.4 Key Challenges facing Software Engineering

Coping with legacy systems, coping with increasing diversity and coping with demands for reduced delivery times:

- ***Legacy systems:*** Old, valuable systems must be maintained and updated
- ***Heterogeneity:*** Systems are distributed and include a mix of hardware and software
- ***Delivery:*** There is increasing pressure for faster delivery of software.

Professional and Ethical Responsibility

- Software engineering involves wider responsibilities than simply the application of technical skills.
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals.
- Ethical behavior is more than simply upholding the law.

Notes

Issues of Professional Responsibility

- ***Confidentiality:*** Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- ***Competence:*** Engineers should not misrepresent their level of competence. They should not knowingly accept work, which is out with, their competence.
- ***Intellectual property rights:*** Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
- ***Computer misuse:*** Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

Task Suggest four attributes, which all software products have.

1.5.5 ACM/IEEE Code of Ethics

The professional societies in the US have cooperated to produce a code of ethical practice.

Members of these organizations sign up to the code of practice when they join.

The code contains eight principles related to the behavior of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

Code of Ethics – Preamble

- The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.
- Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:
  - *Public:* Software engineers shall act consistently with the public interest.
  - *Clients and Employer:* Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
  - *Product:* Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
  - *Judgment:* Software engineers shall maintain integrity and independence in their professional judgment.
  - *Management:* Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

Notes

- *Profession:* Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
- *Colleagues:* Software engineers shall be fair to and supportive of their colleagues.
- *Self:* Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Ethical Dilemmas

- Disagreement in principle with the policies of senior management
- Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system
- Participation in the development of military weapons systems or nuclear systems.

1.5.6 Difference between Software Engineering and Computer Science

Computer science covers the core concepts and technologies involved with how to make a computer do something while software engineering focuses on how to design and build software. The difference seems minimal but there is a major difference in the sense that in Software Engineering you will learn how to analyze, design, build and maintain software in teams. You will learn about working with people (communication, management, and working with non-technical customers), processes for developing software, and how to measure and analyze the software product and the software process.

Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.

Computer science theories are currently insufficient to act as a complete underpinning for software engineering.

1.5.7 Difference between Software Engineering and System Engineering

As mentioned before software engineering deals with building and maintaining software systems. System engineering is an interdisciplinary field of engineering that focuses on the development and organization of complex artificial systems. System engineering integrates other disciplines and specialty groups into a team effort, forming a structured development process that proceeds from concept to production to operation and disposable.

Notes System Engineering considers both the business and technical needs of all customers, with the goal of providing a quality product that meets the user needs.

1.5.8 State of the Practice

Software is an important part of almost every aspect of human endeavor today. Software engineering methods and technology have advance greatly. Professionals with software engineering knowledge and skills are in high demand. Unfortunately, serious problems still need to be addressed.

- Problems with scheduled

Notes

- Problems with cost
- Problems with quality.

Software Development Problems

- Software is becoming larger and more complex, e.g. the code in consumer products is doubling every couple of years.
- Many software products have poor quality, e.g. Software testing typically finds 25 defects per 1000 lines of code.
- Software development is costly, e.g. Development of the FAA's Advance Automation System cost over $700 per line of code.
- Serious scheduled overruns are common, e.g. One in four large-scale development projects is never completed.

Challenges to the Profession

- Serious problems in software development associated with cost, quality, and scheduled still remain.
- Although improving, the software engineering profession is still viewed as immature; and many organizations still have ad hoc software development process.
- The initial professional education of engineers is critical to the advancement of the discipline.

Self Assessment

Fill in the blanks:

11. Software Engineering has a …….. layered framework.
12. Software systems which are intended to provide …….. support for software process activities.
13. Software engineering is concerned with …….. software development.
14. Software is an important part of almost every aspect of ……. endeavor today.

1.6 Software Development Issues

People Issues

a. ***Client/Customer View:*** What do the clients and the customers expect from the software.

b. ***The user viewpoint:*** The users and the client/customers can be different, what do the users expect out of the software.

- ❖ Project Teams
- ❖ Team building
- ❖ Technical competencies
- ❖ Communication

Notes

Figure 1.7: Software Development Roles and Processes

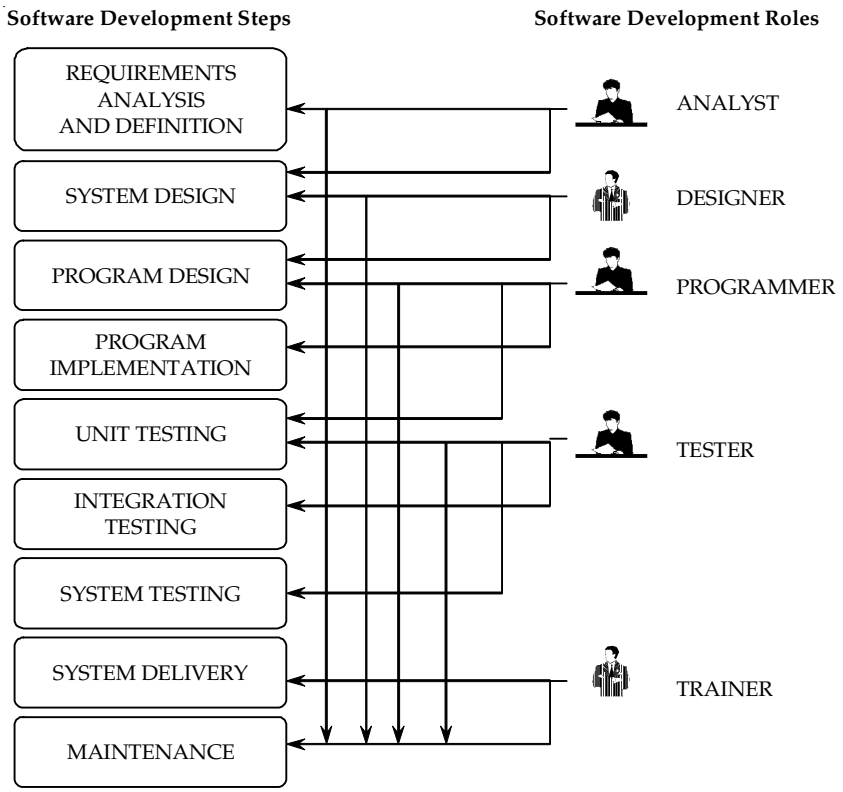


Table 1.1: Wheeler's Most Important Software Innovations

Year	Innovation	Innovator
1945	Stored-program	John von Neumann
1945	Hypertext	Vannevar Bush
1951	Subroutines	Maurice Wilkes, Stanley Gill, and David Wheeler
1952	Compilers	Grace Hopper
1954	Practically Compiling Human -like Notation (FORTRAN)	John B ackus
1957	Time-sharing	John McCarthy
1966	Structured Programming	Böhm and Jacopini
1967	Object Oriented Programming	Simula 67 - Graham M. Birtwistle, Ole - Johan Dahl, Bjoern Myhrhaug, and Kristen Nygaard
1968	The Graphical User Interface (GUI)	Doug las C. Engelbart
1972	Modularity Criteria	David Parnas
1988	Model View Controller	Krasner and Pope
1991	Design Patterns	Erich Gamma
1993	Refactoring	William F. Opdyke

Notes

Self Assessment

Fill in the blanks:

15. Many software companies and developers, they are given a small window of time and a small ..... to release a software package.

Caselet

Software majors slump on NSE

Coimbatore, Nov 3: With the European crisis looming large, the four software majors — Wipro, Infosys, TCS and HCL Tech — were trading in the negative territory on the NSE this morning, just ahead of the opening of the European markets.

The fact that only 15 stocks had gained against 35 that were down indicated the overall market mood with the Nifty down by about 44 points and Sensex by about 140 points.

In terms of value, Infosys was the biggest loser, probably because of its higher price, with the scrip losing ₹ 45.60 to trade around ₹ 2,788. While TCS was down by ₹ 14.10 at ₹ 1,092.75, HCL Tech lost ₹ 4.30 to trade around ₹ 434 and Wipro was down by ₹ 5.60 to trade around ₹ 368.

The other major losers were Hero Motocorp that lost ₹ 33.40 to trade around ₹ 2,055 and Axis Bank which was down by ₹ 21.75 to trade around ₹ 1,104.

Though the gainers' list was long, other than BHEL, which was up by ₹ 7.30 to trade around ₹ 323, GAIL that was up by ₹ 3.80 to quote ₹ 423 and Maruti that was up by ₹ 6.80 to trade at ₹ 1,130.80, most of the others had gained by less than ₹ 1.

1.7 Summary

- Software has become a key element in the evolution of computer-based systems.
- Over the past five decades it has evolved from a specific problem solving and analysis tool to a complete industry in itself. However, some initial stage practices have lead to certain problems today.
- Software comprises of programs, data and documents.
- The purpose of software engineering is to provide a framework to develop high quality software.
- Software engineering is an engineering discipline, which is concerned with all aspects of software production.
- Software products consist of developed programs and associated documentation. Essential product attributes are maintainability, dependability, efficiency and usability.
- The software process consists of activities, which are involved, in developing software products. Basic activities are software specification, development, validation and evolution.
- Methods are organized ways of producing software.
- They include suggestions for the process to be followed, the notations to be used, and rules governing the system descriptions, which are produced and design guidelines.

Notes

- CASE tools are software systems, which are designed to support routine activities in the software process such as editing design diagrams, checking diagram consistency and keeping track of program tests which have been run.

1.8 Keywords

CASE: Computer Aided Software Engineering

CMM: Capability Maturity Model

DOS: Disk Operating System

HR: Human Resources

IPR: Intellectual Property Rights

KPA: Key Process Area

PSP: Personal Software Process

SPICE: Software Process Improvement and Capability Determination

TSP: Team Software Process

1.9 Review Questions

1. The myths mentioned in the section 1.3 are slowly fading away. Try to add one or two new myths to each category.
2. Choose an application and discuss (i) the software application category that is belongs to, (ii) the data contents associated with it and (iii) the information determinacy of the application.
3. "Software is designed and built so that it can be reused in different programs." Substantiate with suitable examples.
4. Suppose you are the software engineer of a modern and technically equipped company then explain how software delivers the most important product of our time—information.
5. Critically analyze the role of computer software. "Software has undergone significant change over a time span of little more than 50 years." Comment.
6. "The software differs from hardware as it is more logical in nature and hence, the difference in characteristics." Discuss.
7. Software is easy to change. It is myth? Explain why or why not? Explain with example.
8. Process defines a framework for a set of Key Process Areas (KPA) that must be established for effective delivery of software engineering technology. Analyze this statement.
9. "Software projects have difficulties to respect initial budget and schedule, which is confirmed by real life experience." Explain.
10. What do you think the biggest problem is with software development?
11. Apart from the challenges of legacy systems, heterogeneity and rapid delivery, identify other problems and challenges that software engineering is likely to face in the 21st century.

Notes

12. Software engineering methods only became widely used when CASE technology became available to support them. Suggest five types of method support, which can be provided by CASE tools.

Answers: Self Assessment

1.	local context	2.	high
3.	information	4.	timing
5.	operating	6.	curve
7.	mechanistic	8.	change
9.	objectives	10.	hardware
11.	three	12.	automated
13.	cost-effective	14.	human
15.	budget		

1.10 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering*, 2nd Edition, PHI.

Richard Fairpy, Software Engineering Concepts, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering—A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering*, 6th Edition, Pearson Education.

Online links

http://www.scribd.com/doc/10880744/Ethical-Issues-in-Software-Development

http://en.wikipedia.org/wiki/Software_engineering

Notes

Unit 2: A Generic View of Process

CONTENTS

Objectives

Introduction

2.1 Software Engineering – A Layered Technology

2.2 A Process Framework

2.2.1 What is a Software Process?

2.2.2 Characteristics of a Software Process

2.2.3 Process Improvement

2.2.4 Software Development Life Cycle (SDLC)

2.2.5 A Process Step Specification

2.3 Capability Maturity Model

2.4 Summary

2.5 Keywords

2.6 Review Questions

2.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize Software Engineering-A layered technology
- Describe process framework
- Demonstrate the capability maturity model integration
- Explain process patterns
- Discuss process assessment

Introduction

Construction of a system, specially a software system, is quite an involved activity. From the inception of an idea for a software system, until it is implemented and delivered to a customer, and even after that, the system undergoes gradual development and evolution. The software is said to have a life cycle composed of several phases. Each of these phases results in the development of either a part of the system or something associated with the system, such as a test plan or a user manual.

A number of process models have been proposed for developing software. In fact, each organization that follows a process has its own customized version. Different processes can also be divided into different activities and activities into sub-activities. However, in general, any problem solving in software must consist of such activities as:

1. Requirement specification for understanding and clearly stating the problem

Notes

2. Deciding a plan for a solution
3. Coding for implementing the planned solution, and
4. Testing for verifying the programs.

These activities may not be done explicitly in small size projects. The start and finish boundaries of these activities may not be clearly defined, and no written record of the activities may be kept. However, for large systems, where the problem-solving activity may last a couple of years and where many people are involved in development, performing these activities implicitly without proper documentation and representation will clearly not work, and each of these four problem solving activities has to be done formally.

Caution In fact, for large systems, each activity can itself be extremely complex, and methodologies and procedures are needed to perform it efficiently and correctly. Each of these activities is a major task for large software projects.

2.1 Software Engineering – A Layered Technology

Software Engineering can be viewed as a layered technology. The main layers are:

- ***Process layer:*** It is an adhesive that enables rational and timely development of computer software. It defines an outline for a set of key process areas that must be acclaimed for effective delivery of software engineering technology.
- ***Method layer:*** It provides technical knowledge for developing software. This layer covers a broad array of tasks that include requirements, analysis, design, program construction, testing and support phases of the software development.
- ***Tools layer:*** It provides computerized or semi-computerized support for the process and method layer. Sometimes, tools are integrated in such a way that other tools can use the information created by one tool. This multi-usage is commonly referred to as computer-aided software engineering or CASE. CASE combines software, hardware and software engineering database to create software engineering analogous to computer-aided design or CAD for hardware. CASE helps in application development including analysis, design, code generation, and debugging and testing etc. This is possible by using CASE tools which provides automated methods for designing and documenting traditional structured programming techniques. The two prominent delivered technologies using case tools are application generators and PC-based workstations that provide graphics-oriented automation of the development process.

The main goal of software engineering is to help developers obtain high-quality software at low cost and with a small time. Therefore, like all other engineering disciplines, software engineering is also driven by three main parameters, namely cost, schedule and quality.

Notes Among these, cost and quality are considered as the primary driving factors in software engineering because the schedule value is more or less fixed for a given cost.

Notes

Self Assessment

Fill in the blanks:

1. Different processes can also be divided into different activities and activities into……..
2. Case combines software, hardware and …….. database to create software engineering analogous to computer-aided design or CAD for hardware.
3. The two prominent delivered technologies using case tools are ……. and PC-based workstations.

2.2 A Process Framework

Software engineering approach pivots around the concept of process. A process means "a particular method of doing something, generally involving a number of steps or operations." In software engineering, the phrase software process refers to the method of developing software.

Software process specifies how one can manage and plan a software development project taking constraints and boundaries into consideration. A software process is a set of activities, together with ordering constraints among them, such that if the activities are performed properly and in accordance with the ordering constraints, the desired result is produced. The desired result is high-quality software at low cost. Clearly, if a process does not scale up and cannot handle large software projects or cannot produce good-quality software, it is not a suitable process.

Major software development organizations typically have many processes executing simultaneously. Many of these do not concern software engineering, though they do impact software development. These could be considered non-software engineering process models. Business process models, social process models, and training models, are all examples of processes that come under this. These processes also affect the software development activity but are beyond the purview of software engineering.

The process that deals with the technical and management issues of software development is called a software process. Clearly, many different types of activities need to be performed to develop software. As we have seen earlier, a software development project must have at least development activities and project management activities. All these activities together comprise the software process.

As different type of activities are being performed, which are frequently done by different people, it is better to view the software process as consisting of many in component processes, each consisting of a certain type of activity. Each of these component processes typically has a different objective, though these processes obviously cooperate with each other to satisfy the overall software engineering objective.

Did u know? **What did the software process framework does?**

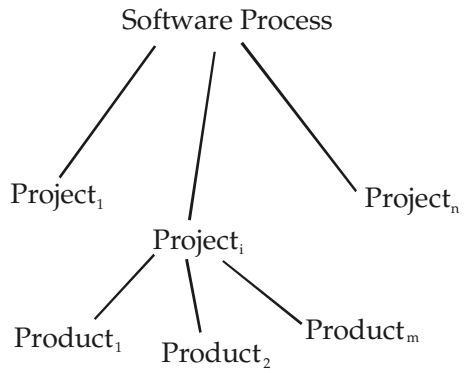
Software Process framework is a set of guidelines, concepts and best practices that describes high level processes in software engineering. It does not talk about how these processes are carried out and in what order.

A software process, as mentioned earlier, specifies a method of developing software. A software project, on the other hand, is a development project in which a software process is used. Software products are the outcomes of a software project. Each software development project starts with some needs and is expected to end with some software that satisfies those needs. A software

Notes

process specifies the abstract set of activities that should be performed to go from user needs to the final product. The actual act of executing the activities for some specific user needs is a software project. And all the outputs that are produced while the activities are being executed are the products. One can view the software process as an abstract type, and each project is done using that process as an instance of this type. In other words, there can be many projects for a process and there can be many products produced in a project. This relationship is shown in Figure 2.1.

Figure 2.1: Relation between Processes, Projects and Products



The sequence of activities specified by the process is typically at an abstract level because they have to be usable for a wide range of projects. Hence, "implementing" them in a project is not straightforward. To clarify this, let us take the example of book writing. A process for book writing on a subject will be something like this:

1. Set objectives of the book – audience, marketing etc.
2. Outline the contents.
3. Research the topics covered.
4. Do literature survey.
5. Write and/or compile the content.
6. Get the content evaluated by a team of experts.
7. Proof read the book.
8. Make corrections, if any.
9. Get the book typeset.
10. Print the book.
11. Bind the book.

Overall, the process specifies activities at an abstract level that are not project- specific.

It is a generic set of activities that does not provide a detailed roadmap for a particular project. The detailed roadmap for a particular project is the project plan that specifies what specific activities to perform for this particular project, when, and how to ensure that the project progresses smoothly. In our book writing example, the project plan to write a book on Software Engineering will be the detailed marked map showing the activities, with other details like plans for getting illustrations, photographs etc.

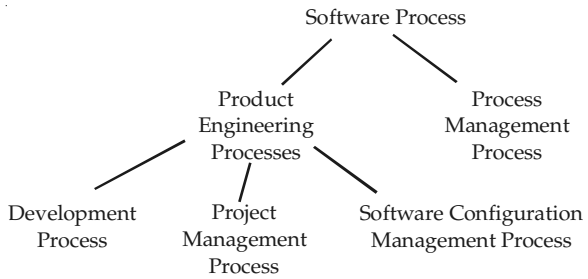
It should be clear that it is the process that drives a project. A process limits the degrees of freedom for a project, by specifying what types of activities must be done and in what order.

Notes

Further, restriction on the degrees of freedom for a particular project are specified by the project plan, which, in itself, is within the boundaries established by the process. In other words, a project plan cannot include performing an activity that is not there in the process.

It is essentially the process that determines the expected outcomes of a project as each project is an instance of the process it follows. Due to this, the focus software engineering is heavily on the process.

Figure 2.2: Software Processes



2.2.1 What is a Software Process?

A software process is the set of activities and associated results that produce a software product. Software engineers mostly carry out these activities. There are four fundamental process activities, which are common to all software processes. These activities are:

1. ***Software specification:*** The functionality of the software and constraints on its operation must be defined.
2. ***Software development:*** The software to meet the specification must be produced.
3. ***Software validation:*** The software must be validated to ensure that it does what the customer wants.
4. ***Software evolution:*** The software must evolve to meet changing customer needs.

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. Different organizations may use different processes to produce the same type of product. However, some processes are more suitable than others for some types of application.

Notes If an inappropriate process is used, this will probably reduce the quality or the usefulness of the software product to be developed.

The Software Process Model

A software process model is a simplified description of a software process, which is presented from a particular perspective. Models, by their very nature, are simplifications, so a software process model is an abstraction of the actual process, which is being described. Process models may include activities, which are part of the software process, software products and the roles of people involved in software engineering. Some examples of the types of software process model that may be produced are:

1. ***A workflow model:*** This shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in this model represent human actions.

Notes

2. ***A dataflow or activity model:*** This represents the process as a set of activities each of which carries out some data transformation. It shows how the input to the process such as a specification is transformed to an output such as a design. The activities here may be at a lower level than activities in a workflow model. They may represent transformations carried out by people or by computers.
3. ***A role/action model:*** This represents the roles of the people involved in the software process and the activities for which they are responsible.

There are a number of different general models or paradigms of software development:

- ***The waterfall approach:*** This takes the above activities and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on. After each stage is defined it is 'signed off' and development goes on to the following stage.
- ***Evolutionary development:*** This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system which satisfies the customer's needs. The system may then be delivered. Alternatively, it may be re-implemented using a more structured approach to produce a more robust and maintainable system.
- ***Formal transformation:*** This approach is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods to a program. These transformations are 'correctness preserving'. This means that you can be sure that the developed program meets its specification.
- ***System assembly from reusable components:*** This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts rather than developing them from scratch.

2.2.2 Characteristics of a Software Process

The fundamental objectives of a process are the same as that of software engineering, namely, optimality and scalability. Optimality means that the process should be able to produce high-quality software at low cost, and scalability means that it should also be applicable for large software projects. To achieve these objectives, a process should have some properties. Some of the important ones are discussed below.

Predictability

Predictability of a process determines how accurately the outcome of following a process in a project can be predicted, before the project is completed. Predictability can be considered a fundamental property of any process. In fact, if a process is not predictable, it is of limited use.

Effective project management is essential for the success of a project, and effective project management revolves around the project plan. A project plan typically contains cost and schedule estimates for the project, along with plans for quality assurance and other activities. Any estimation about a project is based on the properties of the project, and the capability or past experience of the organization.

Example: A simple way of estimating cost could be to say, "this project A is very similar to the project B that we did 3 years ago, hence A's cost will be very close to B's cost." However, even this simple method implies that the process that will be used to develop project A will be

Notes

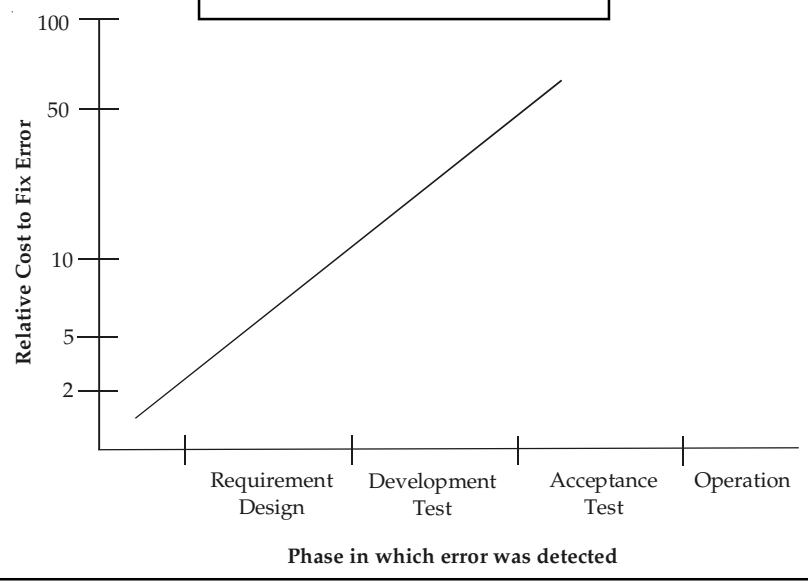
same as the process used for project B, and the process is such, that following the process the second time will produce similar results as the first time. That is, this assumes that the process is predictable. If it was not predictable, then there is no guarantee that doing a similar project the second time using the process will incur a similar cost.

It should be clear that if we want to use past experience to control costs and ensure quality, we must use a process that is predictable. With low predictability, the experience gained through projects is of little value. A predictable process is also said to be under statistical control. A process is under statistical control if following the same process produces similar results.

The crux of this is that we should attempt to detect errors that occur in a phase during that phase itself and should not wait until testing to detect errors. This is not often practiced. In reality, sometimes testing is the only stage where errors are detected. Besides the cost factor, reliance on testing as the primary source for error detection, due to the limitations of testing, will also result in unreliable software. Error detection and correction should be a continuous process that is done throughout software development. In terms of the development phases, this means that we should try to verify the output of each phase before starting with the next.

Detecting errors soon after they have been introduced is clearly an objective that should be supported by the process. However, even better is to provide support for defect prevention. It is generally agreed that all the defect removal methods that exist today are limited in their capability and cannot detect all the defects that are introduced. Furthermore, the cost of defect removal is generally high, particularly if they are not detected for a long time. Clearly, then, to reduce the total number of residual defects that exist in a system at the time of delivery and to reduce the cost of defect removal, an obvious approach is to prevent defects from getting introduced. This requires that the process of performing the activities should be such that fewer defects are introduced. The method, generally, followed to support defect prevention is to use the development process to learn so that the methods of performing activities can be improved.

Figure 2.3: Cost of Correcting Errors



2.2.3 Process Improvement

Process is also not a static entity; it means one has to regularly improve the process according to the need. Improving the quality and reducing the cost of products are fundamental goals of any engineering discipline.

Notes

In the context of software, as the productivity and the cost of a project as well as quality are determined largely by the process, to satisfy the engineering objectives of quality improvement and cost reduction, the software process must be improved.

Having process improvement as a basic goal of the software process implies that the software process used is such that it supports its improvement. This requires that there be means for evaluating the existing process and understanding the weaknesses in the process. Only when support for these activities is available, can process improvement be undertaken. And, as in any evaluation, it is always preferable to have a quantifiable evaluation rather than a subjective evaluation. Hence, it is important that the process provides data that can be used to evaluate the current process and its weaknesses.

2.2.4 Software Development Life Cycle (SDLC)

Software development process is divided into a number of phases. A software is conceived, born, developed, put into operation, maintained during operation and ultimately dies giving way to another software. These phases are aptly dubbed as software development life cycle or SDLC for short.

In the software development process, we have to focus on the activities directly related to production of the software, for example, design, coding, and testing. A development process model specifies some activities that, according to the model, should be performed, and the order in which they should be performed. As stated earlier, for cost, quality, and project management reasons, development processes are generally phased.

As the development process specifies the major development and quality assurance activities that need to be performed in the project, the development process really forms the core of the software process. The management process is decided, based on the development process. Due to the importance of development process, various models have been proposed. In this section, we will discuss some of the major models. As processes consist of a sequence of steps, let us first discuss what should be specified for a step.

Task The management process is decided, based on the development process. In a group of four explain this statement.

2.2.5 A Process Step Specification

A production process is a sequence of steps. Each step performs a well-defined activity leading towards the satisfaction of the project goals, with the output of one step forming the input of the next one. Most process models specify the steps that need to be performed and the order in which they need to be performed. However, when implementing a process model, there are some practical issues, like when to initiate a step and when to terminate a step, that need to be addressed. Here we discuss some of these issues.

As we have seen, a process should aim to detect defects in the phase in which they are introduced. This requires that there be some verification and validation (V&V) at the end of each step. (In verification, consistency with the inputs of the phase is checked, while in validation the consistency with the needs of user is checked.) This implies that there is a clearly defined output of a phase, which can be verified by some means and can form input to the next phase (which may be performed by other people). In other words, it is not acceptable to say that the output of a phase is an idea or a thought in the mind of someone; the output must be a formal and tangible entity. Such outputs of a development process, which are not the final output, are frequently

Notes

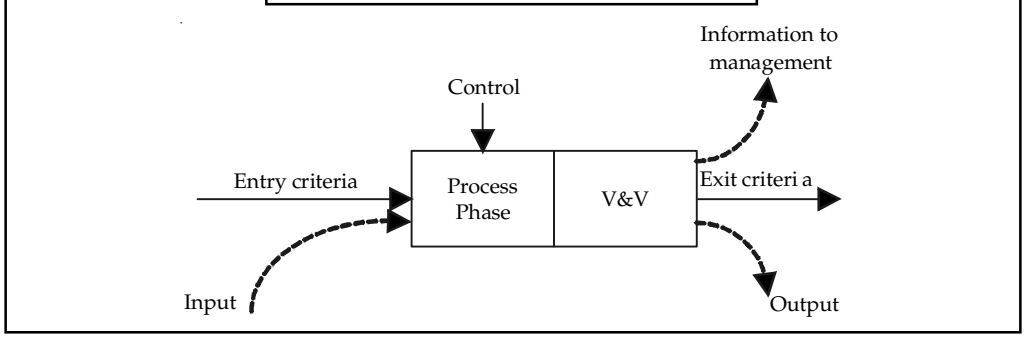
called the work products. In software, a work product can be the requirements document, design document, code, prototype, etc.

This restriction that the output of each step should be some work product, that can be verified, suggests that the process should have a small number of steps. Having too many steps results in too many work products or documents, each requiring V&V, and can be very expensive. Due to this, at the top level, a development process typically consists of a few steps, each satisfying a clear objective and producing a document used for V&V. How to perform the activity of the particular step or phase is generally not an issue of the development process.

As a development process, typically, contains a sequence of steps, the next issue that comes is when a phase should be initiated and terminated. This is frequently done by specifying the entry criteria and exit criteria for a phase. The entry criteria of a phase specify the conditions that the input to the phase should satisfy in order to initiate the activities of that phase. The output criteria specify the conditions that the work product of this phase should satisfy in order to terminate the activities of the phase. The entry and exit criteria specify constraints of when to start and stop an activity. It should be clear that the entry criteria of a phase should be consistent with the exit criteria of the previous phase.

The entry and exit criteria for a phase in a process depend largely on the implementation of the process. For example, for the same process, one organization may have the entry criteria for the design phase as "requirements document signed by the client" and another may have "no more than X errors detected per page in the requirement review." As each phase ends with some V&V activity, a common exit criteria for a phase is "V&V of the phase completed satisfactorily," where satisfactorily is defined by the organization based on the objectives of the project and its experience in using the process. The specification of a step with its input, output, and entry exit criteria is shown in Figure 2.4.

Figure 2.4: A Step in a Development Process



Apart from the entry and exit criteria for the input and output, a development step needs to produce some information for the management process. We know that the basic goal of the management process is to control the development process. For controlling a process, the management process needs to have precise knowledge about the development process activities. As the management process is executed by a logically separate group of people, the information about the development process must flow from the development process to the management process. This requires that a step produce some information for the management process. The management process specifies the nature of this information. This information and other such information from previous steps is used by the management process to exert control on the development process. The flow of information from a step and, exercise of control is also shown in Figure 2.4.

Generally, the information flow from a step is in the form of summary reports describing the amount of resources spent in the phase, schedule information, errors found in the V & V activities, etc. This type of information flow at defined points in the development process makes it possible

Notes

for the project management (i.e., the people executing the management process) to get precise information about the development process, without being directly involved in the development process or without going through the details of all activities of a phase.

Self Assessment

Fill in the blanks:

4. Major software development organizations typically have many processes executing……..
5. A software project, on the other hand, is a …… project in which a software process is used.
6. Software is conceived, born, developed, put into operation, maintained during …….. and ultimately dies giving way to another software.
7. Predictability of a process determines how …… the outcome of a process.

2.3 Capability Maturity Model

The Capability Maturity Model (CMM) is a methodology used to develop and refine an organization's software development process. The model describes a five-level evolutionary path of increasingly organized and systematically more mature processes. CMM was developed and is promoted by the Software Engineering Institute (SEI), a research and development center sponsored by the U.S. Department of Defense (DoD). SEI was founded in 1984 to address software engineering issues and, in a broad sense, to advance software engineering methodologies. More specifically, SEI was established to optimize the process of developing, acquiring, and maintaining heavily software-reliant systems for the DoD. Because the processes involved are equally applicable to the software industry as a whole, SEI advocates industry-wide adoption of the CMM.

Purpose of the CMM

The CMM is a framework that describes the key elements of an effective process. It provides a foundation for process improvement. The CMM describes an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process. The process below describes the CMM. It shows the five levels of progressive process maturity (Initial, Repeatable, Defined, Managed, and Optimizing), and indicates the Process Areas (PA) that are addressed at each level.

The CMM covers practices for planning, engineering, and managing development and maintenance activities. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality. The goal is to improve efficiency, return on investment, and effectiveness.

The CMM establishes a yardstick against which it is possible to judge, in a repeatable way, the maturity of an organization's process and compare it to the state of the practice of the industry. The CMM is also used extensively by organizations to identify process improvement needs, to plan and prioritize improvements, and to evaluate improvement progress.

The CMM has become a de facto industry standard for assessing and improving processes. Through the CMM, the SEI and community have put in place an effective means for modeling, defining, and measuring the maturity of the processes used by process engineering and development professionals. The CMM has been widely adopted and used by the U.S. Government, industry, and academia.

Notes

Figure 2.5: SEI's Software Process Capability Maturity Model

Level	Focus	Key Process Areas	Result
5 Optimizing	Continuous Process Improvement	Defect prevention Technology innovation Process change management	Productivity & Quality
4 Managed	Product and Process Quality	Process measurement and analysis Quality management	
3 Defined	Engineering Process	Organization process focus Organization process definition Peer reviews Training program Intergroup coordination Software product engineering Integrated software management	
2 Repeatable	Project Management	Software project planning Software project tracking Software subcontract mgt. Software quality assurance Software configuration mgt. Requirements management	
1 Initial	Heroes		Risk

CMM's Five Maturity Levels of Software Processes

- At the initial level, processes are disorganized, even chaotic. Success is likely to depend on individual efforts, and is not considered to be repeatable, because processes would not be sufficiently defined and documented to allow them to be replicated.
- At the repeatable level, basic project management techniques are established, and successes could be repeated, because the requisite processes would have been made established, defined, and documented.
- At the defined level, an organization has developed its own standard software process through greater attention to documentation, standardization, and integration.
- At the managed level, an organization monitors and controls its own processes through data collection and analysis.
- At the optimizing level, processes are constantly being improved through monitoring feedback from current processes and introducing innovative processes to better serve the organization's particular needs.

Process Areas by Maturity Level

- ***Level 1-2 (Repeatable)***
  - Configuration Management
  - Quality Assurance
  - Subcontract Management
  - Project Tracking and Oversight
  - Subcontract Management
  - Project Planning
  - Requirements Management

Notes

- *Level 3 (Defined)*
  - Peer Reviews
  - Intergroup Co-ordination
  - Product Engineering
  - Integrated Software Management
  - Training Program
  - Organization Process Definition
  - Organizational Process Focus
- *Level 4 (Managed)*
  - Quality Management
  - Process Measurement and Analysis
- *Level 5 (Optimizing)*
  - Process Change Management
  - Technology Change Management
  - Defect Prevention

Notes Level 1-2 is the disciplined process; Level 2-3 is the standard consistent practice; Level 3-4 is the predictable process; and Level 4-5 is the continuously improving process.

Self Assessment

Fill in the blanks:

8. The CMM covers practices for planning, engineering, and ..... development and maintenance activities.

Case Study

Participative Approach

In this section, we take a look at how two companies made significantly different implementations of the participative approach to software process assessments and in the next section, we present the key lessons learned from these cases.

Company Y

Y is nearly 15 years old, and has grown to become one of the leading consulting companies in Scandinavia. Current customers include large industrial enterprises, government agencies and international organisations. They focus on using an iterative and incremental development method. The company has a flat and open organisation, with a process-oriented structure, and employs about 140 persons. Over 90% of these hold a MSc/MBA.

Contd...

Notes

In the first step (assessment initiation) members from the SPIQ project team had an opening meeting with the manager of technology. This meeting resulted in formulation of two objectives for the assessment:

1. Get a survey of today's process status.
2. Get an "outsiders" view on the company to suggest areas for improvement.

The assessment was mainly focusing on project managers and department leaders. All the questions in the assessment were related to software development. The scope of the assessment in terms of organisational units and issues were all the process-areas of the company. The assessment was decided to be conducted by the SPIQ-team and the manager of technology.

In steps two and three (focus area delineation and criteria development), Y used the standard questionnaire as a starting point for internal discussions, and for the development of tailor-made questionnaires. They did not change anything, which was a bit surprising. The purpose of doing this was the wish for external impulses. No aggregate or composite questions were used; the focus was only on single questions.

In step four (assessment design), the date of the assessment was determined, and it was decided that one of the researchers from SPIQ should hold a presentation for the managers in the company. This was to be followed by the assessment. The presentation was an introduction to SPI with the focus on general improvement. The purpose of this was to describe the questionnaire and its purpose, and also have a quick walkthrough of the questions.

After a short period of planning, all was set for stage five (assessment implementation). After the presentation, the participants (10 persons) from Y filled out the questionnaires in the meeting-room. This gave them the opportunity to discuss and agree upon the interpretation of unclear questions in the questionnaire. The participators in the assessment only answered for the unit and those levels that were part of their own area of responsibility. All the information was treated confidentially. Filling out the questionnaire took about 30 minutes.

In the final step (data analysis and feedback), most of the respondents participated in the analyses and interpretation of the preliminary result. A half-day workshop was set up for this event. The most important results were then presented. The participants raised a lot of questions and issues, and they started a big discussion as they reviewed and evaluated the preliminary findings. Some of the questions were clarified and answered directly, others were answered by reanalysing the data.

The discussion ended in a priority list of four key areas. These were: delivery (time-schedule and budget), customer supplier-relationship, testing and configuration management and risk control. Some of these results were not expected, while others were obvious to the participants.

The next step for company Y will be an internal discussion of the results, and then figure out the necessary SPI actions. However, they are first going to co-ordinate this work with the work going on in a parallel project.

The result of the last section of the questionnaire was also of great interest. This section is divided into tree sub-sections, and is concerned about finding the most important factors enabling SPI success in the organisation. The most important arguments in favour of SPI in the company Y were: Continuous adjustment to external conditions, job satisfaction,

Contd...

Notes

and vital importance for future competitiveness. The most important arguments against SPI were: Increasing workload (resource demanding), SPI suppresses the creativity and the sense of responsibility, and it moves focus away from the project.

The most important factors for ensuring successful process improvement in the company were: Management involvement, motivation/employee participation, and well-defined and simple routines.

Company X

X is one of the leading companies in their field. Their products are a combination of software (embedded software or firmware) and hardware. In addition to corporate offices and manufacturing facilities in Norway, X has significant marketing, sales and support operations in the USA, Europe, and the Far East. The company employs about 550 persons in Norway, of which the firmware division employs 30 persons.

During the first step (assessment initiation), members from the SPIQ project team and company X had an opening meeting where the objectives of the assessment were set. X wanted to get a survey of today's process status and potential for improvements. After identifying some key areas of improvement, the intention was to make a plan for SPI-actions. The assessment was focusing on project managers, customers and co-workers in two firmware departments and one hardware department. These groups were divided into tree subgroups: Managers, customers and developers. All departments in company X were represented. The assessment was decided to be conducted by the SPIQ-team and the people responsible for the assessment at company X.

In steps two and three (focus area delineation and criteria development), X used the standard questionnaire as a starting point for internal discussions, and for the development of a tailor-made questionnaire. A committee was put together for evaluating the questions. In this committee there were two designers, four managers, two customers, and one from the quality department.

After two meetings the number of questions were doubled. None of the original questions were removed. We recommended X to reduce the number of questions, and to do some minor changes in the wording of items, in order to be more precise in the question text. Some rewording was subsequently done. However, the number of questions was not reduced. No aggregate or composite questions were used; the focus was only on analysing answers to single questions.

During step four (assessment design), it was decided that the assessment co-ordinator in the company should hold a presentation regarding the assessments role in improving X's software processes. In step five (assessment implementation), the presentation was hold. Most of the participants in the assessment were present at this meeting. After the presentation they had 30 minutes to fill out the questionnaire. This was too little time, however, so almost everyone had to deliver the questionnaire later. The leader of the assessment personally visited those who did not show up for the presentation helping them to complete the questionnaire. 32 employees from company X participated, however, four did not deliver the forms. The participators in the assessment only answered for the unit and those levels that were part of their own area of responsibility. Most of the respondents participated in the analyses and interpretation of the presented result in step six (data analysis and feedback). A half-day workshop was set up for this event. The most important results were then presented. The problem with this session was the lack of discussion. Although this complicated the process, the session ended with a priority list of four key areas. These were: Flexibility vs. stability, long-term quality, teamwork, and learning from past experiences.

Contd...

Notes

The next step for company X will be an internal discussion of these results, and to start a process to suggest alternative SPI actions to start with. This job is a bit difficult at the moment because company X is in the middle of a rearrangement.

The result of the last section of the questionnaire was also of great interest. The most important arguments in favour of SPI in the firmware-group were: Quality, motivation/ employee participation, and job satisfaction. The most important arguments against SPI were: "This only creates new procedures and rules" and "a waste of recourses (bad priority)."

The most important factors for ensuring successful process improvement in the firmware-group were: Motivation, developer/designer in focus, and management involvement

Discussion of the Cases

These cases are from two quite different companies; Y, a pure software company and X, a combined software and hardware company with their own production. They both had the same method to follow, but the accomplishment was quite different in a lot of areas. The objectives of the assessment was much the same.

Company Y did not work on the questionnaire template, and let the researchers perform the assessment. The questionnaire was therefore not as tailor-made as one would expect. The reason for this was, as explained before, the wish for only external input to the assessment. If this way of conducting the assessment is successful or not is too early to conclude.

On the other hand, company X did a lot of adjustments and therefore developed a highly tailor-made questionnaire. The problem with this case was the lack of involvement from the researcher's side. To many questions were produced without removing any from the template. Too many questions were too similar, and there were problems interpreting some of them.

With this situation in mind, one could expect that there would be a great discussion on the result from the tailor-made questionnaire, and less discussion on the result from the standard questionnaire. It was a big surprise that the opposite occurred. There could be a lot of reasons for this: At company X over 20 persons participated in the discussion, at Y there were only 10 persons. Also, the participants at X were a mixture of managers and developers, and there is a possibility that this prevented people from speaking out.

Another distinction between the two companies is the composition of the groups that participated in the assessment. In company Y, the group was homogenous (only process managers), but in company X there were three different groups. In this kind of assessment, the results are more interesting if there is a large group answering the questions, and if they come from different parts of the companies. This was the case at company X.

Comparing data from different groups and between different members of the same group gave interesting results. For example did Project manager have the opinion that the level in "Current strength" (topic: "making fast design changes") was low and that one should improve this area significantly. The developers had the opposite opinion. They meant the level today was too high, and wanted to decrease it. People from the customers group thought the level was OK.

The results from the discussions had very little in common. The results from the fourth section of the questionnaire had more in common. Under the category "The most important arguments in favour of SPI", the results tell us that both companies think that SPI activities

Contd...

Notes

will improve quality and make the employees more satisfied. SPI activities seem like a necessary thing to do if you want to achieve success.

In the category "The most important arguments against SPI", the companies had the same opinion on SPI-work increasing the workload and as a source of new procedures and rules, which will cost a lot of resources and move the focus away from the projects. Comparing the results from these categories is interesting, because they first argue that SPI is necessary for the company to survive, but there is a lot of negative work to be done doing this. Maybe SPI has a problem with the association of "quality control"!

Under "The most important factors for ensuring successful process improvement", the companies had two factors in common: Employee participation and management involvement.

During the presentation of the data, there were a lot of "expected" conclusions, but also some the company had never thought about. The conclusions they had expected had, however, never been externalized before. This happened for the first time as a result of the assessments.

2.4 Summary

- The main goal of software engineering is to help developers obtain high-quality software at low cost and with a small time.
- Software engineering approach pivots around the concept of process. A process means "a particular method of doing something, generally involving a number of steps or operations."
- In software engineering, the phrase software process refers to the method of developing software.
- The Capability Maturity Model (CMM) is a methodology used to develop and refine an organization's software development process.

2.5 Keywords

CAD: Computer Aided Design

CMM: Capability Maturity Model

DOD: Department of Defense

OOSP: Object-Oriented Software Process

SEI: Software Engineering Institute

SPC: Statistical Process Control

SPI: Software Process Improvement

2.6 Review Questions

1. Software Engineering can be viewed as a layered technology. Explain.
2. Why it is important that the process provides data that can be used to evaluate the current process and its weaknesses.

Notes

3. The main goal of software engineering is to help developers obtain high-quality software at low cost and with a small time. Discuss.
4. Software products are the outcomes of a software project. Comment.
5. Software development process is divided into a number of phases. Describe.
6. A production process is a sequence of steps. Explain.
7. The CMM has been widely adopted and used by the U.S. Government, industry, and academia. Examine.

Self Assessment

1. sub-activities
2. software engineering
3. application generators
4. simultaneously
5. development
6. operation
7. accurately
8. managing

2.7 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering,* 2nd Edition, PHI.

Richard Fairpy, *Software Engineering Concepts,* Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering – A Practitioner's Approach,* 5th Edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering,* 6th Edition, Pearson Education

Online links

http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Presentations/PDF/ch4.pdf

http://en.wikipedia.org/wiki/Process_patterns

Notes

Unit 3: Process Models

CONTENTS

Objectives
Introduction
3.1 Prescriptive Models
3.1.1 Waterfall Model
3.2 Incremental Process Models
3.2.1 Incremental Model
3.2.2 RAD Model
3.3 Summary
3.4 Keywords
3.5 Review Questions
3.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize the perspective model
- Describe the incremental models
- Demonstrate the waterfall model
- Explain the RAD model

Introduction

There are three major models of the software engineering process. The waterfall model was for many years the front runner and is required for many military applications. (The procurement process of the US military has had a profound effect on computer science and software engineering in particular.) It is a rigid system well suited to applications where the buyer is as sophisticated about computer systems as the seller.

Prototyping is popular with those who work in the marketplace of "user friendly" software. This software is historically less maintainable (and less maintained) than the so called "big iron" applications that were historically developed using the waterfall model. The incremental model works from the notion that software is built. The Spiral model is a more recent model that tries to enjoy the best of both worlds. Each of these models shares some common features. In particular they all involve establishing the requirements that the system is to meet.

3.1 Prescriptive Models

Prescriptive process models advocate an orderly approach to software engineering that leads to a few questions

- If prescriptive process models strive for structure and order are they inappropriate for a software world that thrives on change?

Notes

- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

It defines a distinct set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software.

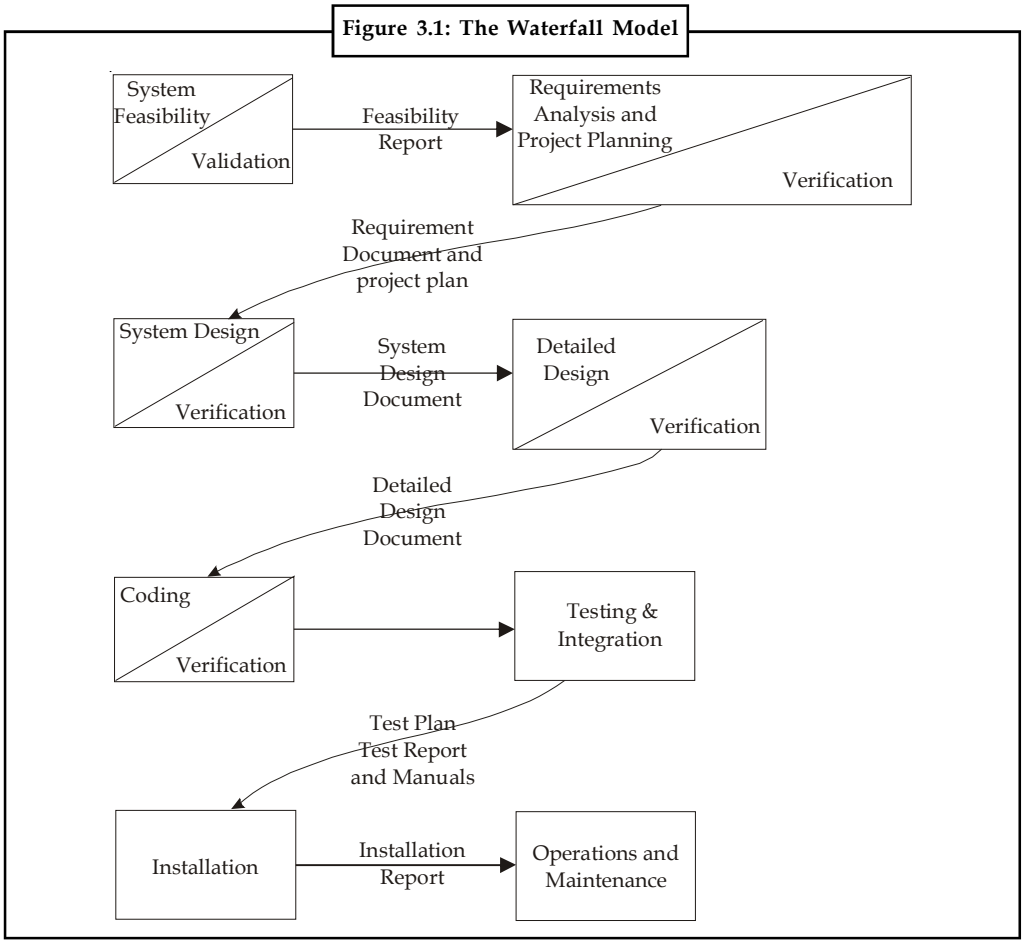
The activities may be waterfall model, incremental, or evolutionary.

Did u know? **What is the exact meaning of perspective modeling?**

A modeling perspective in information systems is a particular way to represent pre-selected aspects of a system. Any perspective has a different focus, conceptualization, dedication and visualization of what the model is representing.

3.1.1 Waterfall Model

Figure 3.1: The Waterfall Model



The simplest software development life cycle model is the waterfall model, which states that the phases are organized in a linear order. However, there are many variations of the waterfall model depending on the nature of activities and the flow of control between them. In a typical model, a project begins with feasibility analysis. On successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins. The design starts after the requirements analysis is complete and coding begins after the design is complete. Once the

Notes

programming is completed, the code is integrated and testing is done. On successful completion of testing, the system is installed. After this, the regular operation and maintenance of the system takes place. The model is shown in Figure 3.2.

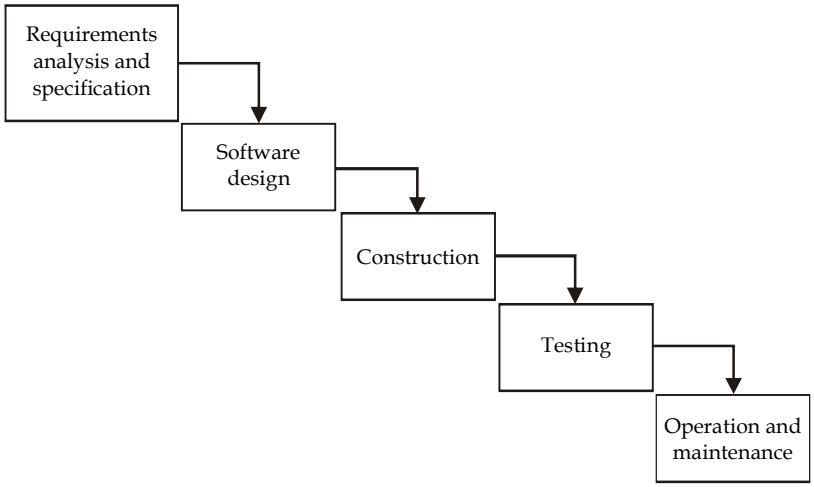
We will discuss the activities related to the development of software. Thus, we will only discuss phases from requirements analysis to testing. The requirements analysis phase is mentioned as "analysis and planning." Planning is a critical activity in software development. A good plan is based on the requirements of the system and should be done before later phases begin. However, in practice, detailed requirements are not necessary for planning. Consequently, planning usually overlaps with the requirements analysis, and a plan is ready before the later phases begin. This plan is an additional input to all the later phases.

Task Explain how waterfall model comes under the perspective modeling? Give examples to support your answer.

With the waterfall model, the sequence of activities performed in a software development project is: requirement analysis, project planning, system design, detailed design, coding and unit testing, system integration and testing. This is the order in which the different phases will be discussed in this book, keeping the sequence as close as possible to the sequence in which the activities are performed.

Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and the beginning of the next, some certification mechanism has to be employed at the end of each phase. This is usually done by some verification and validation means that will ensure that the output of a phase is consistent with its input (which is the output of the previous phase), and that the output of the phase is consistent with the overall requirements of the system.

Figure 3.2: Waterfall Model



The consequence of the need for certification is that each phase must have some defined output that can be evaluated and certified. That is, when the activities of a phase are completed, there should be some product that is produced by that phase. And the goal of a phase is to produce this product. The outputs of the earlier phases are often called work products (or intermediate products) and are usually in the form of documents like the requirements document or design document. For the coding phase, the output is the code. From this point of view, the output of a

Notes

software project is not just the final program along with the user documentation, but also the requirements document, design document, project plan, test plan, and test results.

The waterfall model derives its name due to the cascading effect from one phase to the other as is illustrated in the Figure 3.2. In this model each phase is well defined, has a starting and ending point, with identifiable deliveries to the next phase.

This model has five phases: (i) requirement analysis and specification, (ii) design, (iii) implementation and unit testing, (iv) integration and system testing and (v) operation and maintenance. All the phases occur one-by-one without overlapping. The next phase begins only after the end of the previous phase.

- ***Requirement analysis and specification:*** This phase aims at understanding the exact requirements of the customer and documents them. Both the customer and the software developer work together so as to document all the functions, performance and interfacing requirements of the software. It describes the "what" of the system to be produced and not "how". In this phase a large document called Software Requirement Specification document (SRS) is produced which contains the detailed description of what the system will do in the common language.
- ***Design phase:*** The aim of this phase is to transform the requirements gathered in the SRS into a suitable form which permits further coding in a programming language. It defines the overall software architecture together with high level and detailed design. All this work is documented as Software Design Description document (SDD).
- ***Implementation and unit testing phase:*** The actual coding begins at his stage. The implementation goes smoothly if the SDD has complete information required by the software engineers. During testing, the code is thoroughly examined and modified. Small modules are testes in isolation initially. Thereafter, these modules are tested by writing some overhead code in order to check the interaction between these modules and the flow of intermediate output.
- ***Integration and system testing:*** This phase is highly crucial as the quality of the end product is determined by the effectiveness of the testing carried out. Better output will lead to satisfied customers, lower maintenance costs and accurate results. Unit testing determines the efficiency of individual modules. However, in this phase the modules are tested for their interactions with each other and with the system.
- ***Operation and maintenance phase:*** Maintenance is the task performed by every user once the software has been delivered to the customer, installed and operational.

  Thus, the delivery of software initiates the maintenance phase. The time and efforts spent on the software to keep it operational after release is important. It includes activities like error correction, removal of obsolete functions, optimization and enhancements of functions. It may span for 5 to 50 years.

Notes Thus, in this model the requirements must be clear at the very initial stages. The end product is available relatively late which in turn delays the error recovery.

Let us now consider the rationale behind the waterfall model. There are two basic assumptions for justifying the linear ordering of phases in the manner proposed by the waterfall model.

1. For a successful project resulting in a successful product, all phases listed in the waterfall model must be performed anyway.
2. Any different ordering of the phases will result in a less successful software product.

Notes

A successful software product is one that satisfies all the objectives of the development project. These objectives include satisfying the requirements and performing the development within time and cost constraints. Generally, for any reasonable project size, all the phases listed in the model must be performed explicitly and formally. Informally performing the phases will work only for very small projects. The second reason is the one that is now under debate. For many projects, the linear ordering of these phases is clearly the optimum way to organize these activities. However, some argue that for many projects this ordering of activities is infeasible or sub-optimum. We will discuss some of these ideas shortly. Still, the waterfall model is conceptually the simplest process model for software development that has been used most often.

Project Outputs in Waterfall Model

As we have seen, the output of a project employing the waterfall model is not just the final program along and documentation to use it. There are a number of intermediate outputs that must be produced to produce a successful product. Though the set of documents that should be produced in a project is dependent on how the process is implemented, the following is a set of documents that generally forms the minimum set that should be produced in each project:

- Requirements document
- Project plan
- System design document
- Detailed design document
- Test plan and test reports
- Final code
- Software manuals (e.g., user, installation, etc.)
- Review reports

Except for the last one, these are the outputs of the phases, and they have been briefly discussed. To certify an output product of a phase before the next phase begins, reviews are often held. Reviews are necessary, especially for the requirements and design phases, because other certification means are frequently not available. Reviews are formal meetings to uncover deficiencies in a product and will be discussed in more detail later. The review reports are the outcome of these reviews.

Task What have you understood by the project outputs of waterfall model?

Limitations of the Waterfall Model

Software life-cycle models of the waterfall variety are among the first important attempts to structure the software life cycle. However, the waterfall model has limitations. Like a waterfall, progress flows in one direction only, towards completion of the project (from one phase to the next). Schedule compression relaxes this requirement but introduces new complexities.

Well-defined phases, frozen deliverables, and formal change control make the waterfall model a tightly controlled approach to software development. The waterfall model's success hinges, however, on the ability:

- To set the objectives,

Notes

- To state requirements explicitly,
- To gather all the knowledge necessary for planning the entire project in the beginning.
- The waterfall model's original objectives were to make small, manageable, individual development steps (the phases) and to provide enough control to prevent runaway projects. There are a few problems that have caused dissatisfaction with this phased life-cycle approach.
- The new software system becomes useful only when it is totally finished, which may create problems for cash flow and conflict with organizational (financial) objectives or constraints. Too much money may be tied up during the time the software is developed.
- Neither user nor management can see how good or bad the system is until it comes in. The users may not have a chance to get used to the system gradually.
- Changes that "aren't supposed to happen", are not viewed kindly, neither for requirements nor during the operational life of the system. This can shorten the software's useful life.

Because of these shortcomings, other process models have appeared. They are based on the concept of iteration and evolution.

Waterfall model, although widely used, has some limitations. Here we list some of these:

Despite these limitations, the waterfall model is the most widely used process model. It is well suited for routine types of projects where the requirements are well understood. That is, if the developing organization is quite familiar with the problem domain and the requirements for the software are quite clear, the waterfall model works well.

Self Assessment

Fill in the blanks:

1. The design starts after the requirements analysis is complete and …….. begins after the design is complete.
2. …….. is a critical activity in software development.
3. The consequence of the need for …….. is that each phase must have some defined output that can be evaluated and certified.
4. A successful software product is one that satisfies all the objectives of the ……. project.
5. Schedule compression relaxes this requirement but introduces new……..
6. The waterfall model's original objectives were to make small, ……., individual development steps to provide enough control to prevent runaway projects.
7. Requirement Specification document (SRS) is produced which contains the detailed description of what the system will do in the …… language.

3.2 Incremental Process Models

3.2.1 Incremental Model

The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into

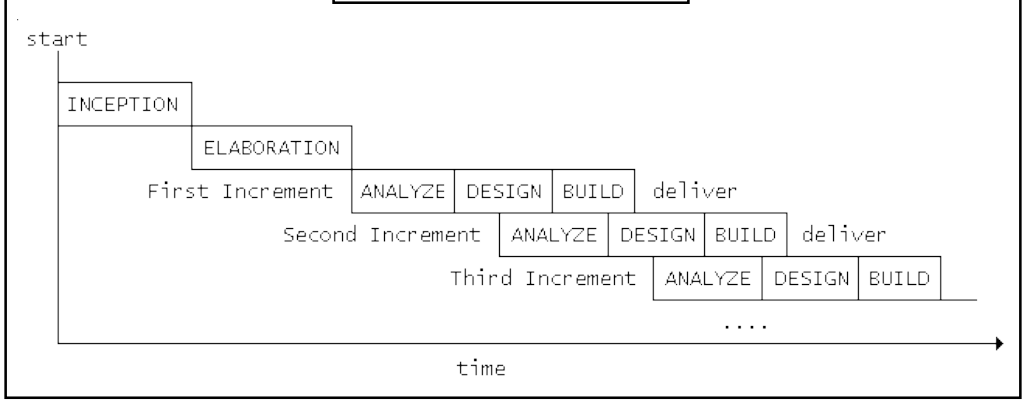
Notes

smaller, more easily managed iterations. Each iteration passes through the requirements, design, implementation and testing phases.

A working version of software is produced during the first iteration, so you have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

- Used when requirements are well understood
- Multiple independent deliveries are identified
- Work flow is in a linear (i.e., sequential) fashion within an increment and is staggered between increments
- Iterative in nature; focuses on an operational product with each increment
- Provides a needed set of functionality sooner while delivering optional components later
- Useful also when staffing is too short for a full-scale development

Figure 3.3: Incremental Model



The incremental approach attempts to combine the waterfall sequence with some of the advantages of prototyping. This approach is favored by many object-oriented practitioners. It basically divides the overall project into a number of increments. Then it applies the waterfall model to each increment. The system is put into production when the first increment is delivered.

Notes The time passes additional increments are completed and added to the working system. This approach is favored by many object-oriented practitioners.

Incremental Model Phases

- ***Inception:*** During the inception phase, the purpose, business rationale, and scope of the project is determined. This is similar to the feasibility analysis that is done in other life cycles.
- ***Elaboration:*** During the elaboration phase, more detailed requirements are collected, high level analysis is performed, and a general architecture is determined. This phase divides the requirements into increments that can be built separately. As we will see, each increment consists of a subset of high level use cases that capture the user's requirements.

Notes

- ***Construction:*** The construction phase builds increments of the system. Each increment is developed using a waterfall approach. This includes detailed analysis and design for the use cases in the increment and coding and testing of the event processors that implement the sequence of events defined by the use cases. The result is production quality software that satisfies a subset of the requirements and is delivered to the end users. Work on different increments may be done in parallel.
- ***Transition:*** The transition phase is the last phase in the project. This may include such things a performance tuning and rollout to all users.

Advantages

- Generates working software quickly and early during the software life cycle.
- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

Disadvantages

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle.

Task Analyze the role of transition phase in incremental model.

3.2.2 RAD Model

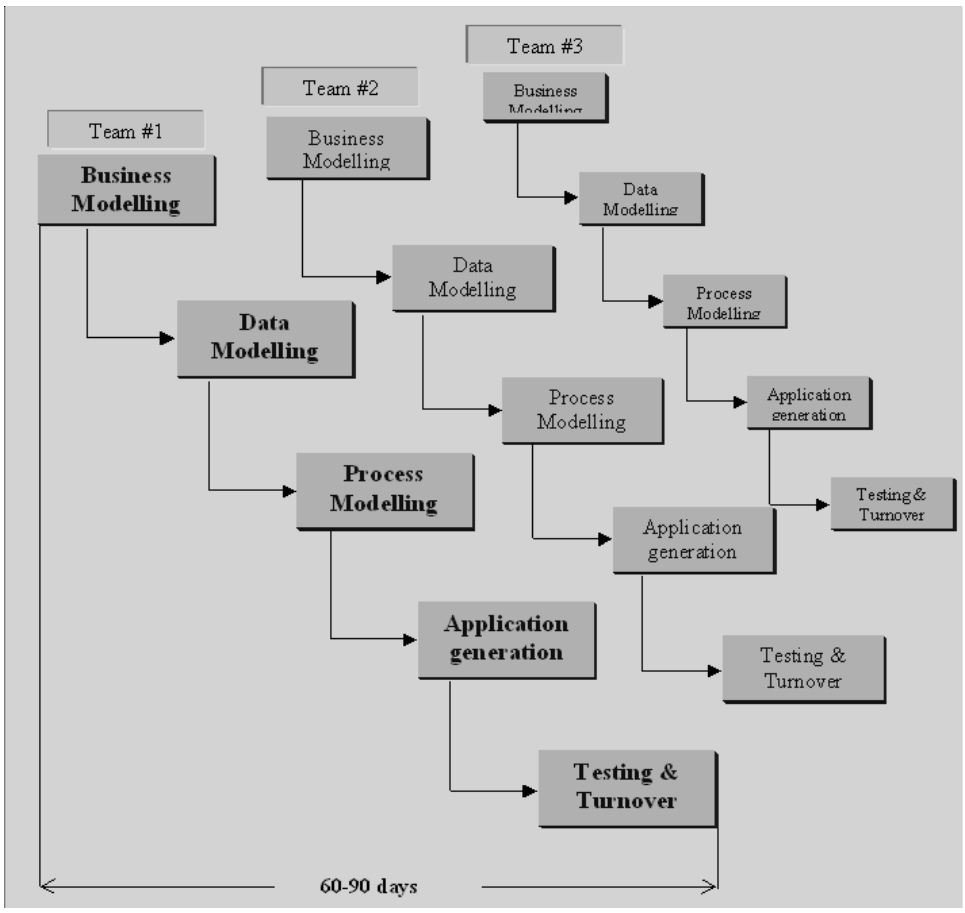
RAD is a linear sequential software development process model that emphasis an extremely short development cycle using a component based construction approach. If the requirements are well understood and defines, and the project scope is constraint, the RAD process enables a development team to create a fully functional system with in very short time period.

RAD (rapid application development) is a concept that products can be developed faster and of higher quality through:

- Gathering requirements using workshops or focus groups
- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that defers design improvements to the next product version
- Less formality in reviews and other team communication

Notes

Figure 3.4: RAD Model



RAD Model Phases

RAD model has the following phases:

- ***Business Modeling:*** The information flow among business functions is defined by answering questions like what information drives the business process, what information is generated, who generates it, where does the information go, who process it and so on.
- ***Data Modeling:*** The information collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified and the relation between these data objects (entities) is defined.
- ***Process Modeling:*** The data object defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting or retrieving a data object.
- ***Application Generation:*** Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.
- ***Testing and Turn over:*** Many of the programming components have already been tested since RAD emphasis reuse. This reduces overall testing time. But new components must be tested and all interfaces must be fully exercised.

Notes

Did u know? **In which programming language the RAD model is commonly use?**

RAD usually embraces object-oriented programming methodology, which inherently fosters software re-use.

Advantages and Disadvantages

RAD reduces the development time and reusability of components help to speed up development. All functions are modularized so it is easy to work with.

For large projects RAD require highly skilled engineers in the team. Both end customer and developer should be committed to complete the system in a much abbreviated time frame. If commitment is lacking RAD will fail. RAD is based on Object Oriented approach and if it is difficult to modularize the project the RAD may not work well.

Self Assessment

Fill in the blanks:

8. A rigidly paced schedule that defers design improvements to the next …….. version
9. The construction phase builds …….. of the system.
10. During the inception phase, the purpose, ……. , and scope of the project is determined.
11. The incremental approach attempts to combine the waterfall sequence with some of the advantages of……...
12. The incremental model is an intuitive approach to the ……. model.
13. Cycles are divided up into smaller, more easily managed …….
14. ……. is a concept that products can be developed faster and of higher quality.
15. ……. is easier to manage risk because risky pieces are identified and handled during its iteration.

Caselet

Indicom Software and Services

Bridging Technology, Education & Industry

We are committed to creating compelling experiences for our customers and partners. We shall inspire technology users and create achievers who would innovate and contribute back to society.

Indicom Software and Services, Chennai, is among the foremost of organizations extending high quality technology solutions to educational institutions. Its passion for excellence and its mission to bridge the gap between industry and education has helped it evolve several IT solutions and services for the benefit of the Academic community.

Contd...

Notes

Its partnership with software majors including Microsoft, Borland, Adobe, Macromedia, PTC and Epiance has been an enriching and fulfilling experience, while extending exclusive and focused technology solutions and programmes for academic institutions and corporates across India. Indicom Software and Services is a highly motivated team, with qualified, experienced and committed executives heading its marketing, support, training and administrative services.

Software and Services

We are premier partners to Microsoft, Borland, Adobe & PTC and have retained the No.1 Academic Partner position of Microsoft for the last five years. Our expertise in customizing, deploying and training on these software is our USP. We have deployed over 50 of Microsoft's official .NET mobile computing centres in premier institutions across India.

E-learning & Content Development

Our e-learning courses range from programming using .NET to personal productivity using Office. We also have the capability of custom-building exclusive content based on specific requirements. These modules include testing, grading and analyzing tools to create a complete learning experience.

Training

Our team of corporate trainers consists of experienced software professionals with a high level of skill set in a variety of technologies including Microsoft .NET, J2EE and the entire range of Microsoft server software. Our training programmes are designed to equip freshers and also to aid existing professionals in technology migration and upgradation.

TechCamp

Tech Camp is an absolutely unique skills development program that will make students instantly ready for the IT industry. Intensive training will be immediately followed by a REAL LIFE PROJECT supported by industry. The TechCamp model is designed to equip you with exactly what the industry is looking for.

Business Process Improvement

We provide a unique, data-driven approach to process intelligence to increase employee productivity. Starting with a capture agent that provides detailed application audit information from every end-user action, the software delivers comprehensive insight into process execution, including detailed individual tasks and activities. This technology can help in:

1. Cutting down cost of training and retraining employees
2. Identifying error prone and difficult tasks
3. Improving existing processes

Our Clients

Our clients include Corporate Organisations (Microsoft, Savvy Soft, MGM Group, Parkin's, Sutherland, Hardy Exploration, Ashok Leyland, Schlumburger, Aban, Greeva) and Academic Institutions (Anna University, IIT Madras, Madras University, SRM University, Dr. MGR University, VIT, SASTRA), Hindustan group, Dr. Mahalingam group, Thiagarajar, NIOT, PSG, CIT, RMK group.

Notes

3.3 Summary

- The simplest software development life cycle model is the waterfall model, which states that the phases are organized in a linear order.
- There are many variations of the waterfall model depending on the nature of activities and the flow of control between them. In a typical model, a project begins with feasibility analysis. On successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins.
- In incremental model, a series of releases called 'increments' are delivered that provide more functionality progressively for customer as each increment is delivered.
- The first increment is known as core product. This core product is used by customers and a plan is developed for next increment and modifications are made to meet the needs of the customer. The process is repeated.
- If the requirements are well understood and defines, and the project scope is constraint, the RAD process enables a development team to create a fully functional system with in very short time period.

3.4 Keywords

RAD: Rapid Application Development

SRS: Requirement Specification document

SDD: Software Design Description document

3.5 Review Questions

1. The simplest software development life cycle model is the waterfall model, which states that the phases are organized in a linear order. Justify this statement with diagram.
2. The requirements analysis phase is mentioned as "analysis and planning." What have you understood from this statement?
3. "A good plan is based on the requirements of the system and should be done before later phases begin". Discuss.
4. Why RAD reduces the development time and reusability of components help to speed up development? Explain.
5. Discuss why RAD is a linear sequential software development process model that emphasis an extremely short development cycle using a component based construction approach?
6. Critically analyze the various RAD Model Phases.
7. Briefly explain the incremental phases of the incremental model.
8. A working version of software is produced during the first iteration, so you have working software early on during the software life cycle. Comment.
9. Is waterfall model perspective model or incremental model? Explain with examples.
10. Subsequent iterations build on the initial software produced during the first iteration. Why or why not? Justify your answer.

Notes

Answers: Self Assessment

1. coding
2. Planning
3. certification
4. development
5. complexities
6. manageable
7. common
8. product
9. increments
10. business rationale
11. prototyping
12. waterfall
13. iterations
14. RAD (rapid application development)
15. Incremental model

3.6 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering,* 2nd Edition, PHI.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher Education.

Sommerville, *Software Engineering*, 6th Edition, Pearson Education.

Online links

http://productdevelop.blogspot.com/2011/07/what-is-incremental-model-in-software.html

http://en.wikipedia.org/wiki/Rapid_application_development

http://www.waterfall-model.com/

http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf

Notes

Unit 4: Evolutionary Process Models

CONTENTS

Objectives

Introduction

4.1 Evolutionary Process Model

4.1.1 Benefits of Evolutionary Development Model

4.2 Prototyping Model

4.3 The Spiral Model

4.4 Concurrent Development Model

4.5 A Final Comment on Evolutionary Models

4.6 Summary

4.7 Keywords

4.8 Review Questions

4.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Demonstrate evolutionary process models
- Recognize Prototyping and Spiral Models
- Describe the concurrent development model

Introduction

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; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

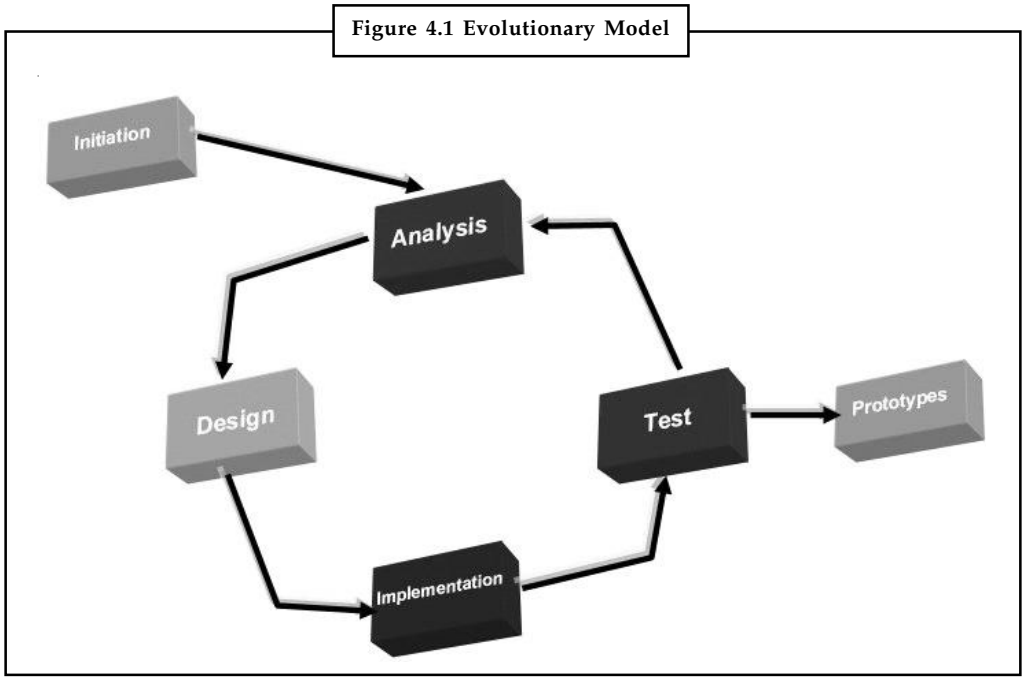
The linear sequential model is designed for straight-line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The prototyping model is designed to assist the customer (or developer) in understanding requirements. In general, it is not designed to deliver a production system. The evolutionary nature of software is not considered in either of these classic software engineering paradigms.

Notes Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

4.1 Evolutionary Process Model

Notes

Figure 4.1 Evolutionary Model



The evolutionary model could be seen as one of the classic iterative activity based models. Each of the activities are handled in sequential order. After each of the iterations an evolutionary prototype is produced. This prototype is used to validate the iteration it's just coming from and forms the basis of the requirements for the next phase. It's usually applied when the requirements are unclear. There are two well-known forms: exploratory programming and prototyping.

Did u know? **Who is Theodosious Dobzhansky?**

Theodosious Dobzhansky defined evolution as the change in allele frequencies over time, a definition that appears to be more applicable to fine-scale population genetics questions than, necessarily, to between-species evolutionary relationships.

A prototype is used to discover the requirements and is discarded once the real system is developed. In exploratory programming a system is implemented and functionality is added iteratively.

One of the major problems here is the way of dealing with the development of software. The prototypes produced are being used in the next phase. Because many prototypes are produced that are redefined further and further, the whole software might become a patchwork of features. Hence the whole approach might only work for small projects.

The waterfall model is viable for software products that do not change very much once they are specified. But for software products that have their feature sets redefined during development because of user feedback and other factors, the traditional waterfall model is no longer appropriate.

- The Evolutionary EVO development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle.

Notes

- Feedback is provided by the users on the product for the planning stage of the next cycle and the development team responds, often by changing the product, plans, or process.
- These incremental cycles are typically two to four weeks in duration and continue until the product is shipped.

4.1.1 Benefits of Evolutionary Development Model

- Benefit not only business results but marketing and internal operations as well.
- Use of EVO brings significant reduction in risk for software projects.
- EVO can reduce costs by providing a structured, disciplined avenue for experimentation.
- EVO allows the marketing department access to early deliveries, facilitating development of documentation and demonstrations.
- Short, frequent EVO cycles have some distinct advantages for internal processes and people considerations.
- The cooperation and flexibility required by EVO of each developer results in greater teamwork.
- Better fit the product to user needs and market requirements.
- Manage project risk with definition of early cycle content.
- Uncover key issues early and focus attention appropriately.
- Increase the opportunity to hit market windows.
- Accelerate sales cycles with early customer exposure.
- Increase management visibility of project progress.
- Increase product team productivity and motivation.

Caution Large or complex projects can't be handled this way as the software might become too hard to manage. Evolutionary process model consist of prototyping and spiral model.

Self Assessment

Fill in the blanks:

1. A process model that views development as a series of hills, each representing a separate loop of the …… model.
2. Use of evolutionary model brings significant …….. in risk for software projects.
3. The …….. and flexibility required by evolutionary model of each developer results in greater teamwork.
4. Feedback is provided by the users on the …… for the planning stage of the next cycle and the development team responds, often by changing the product, plans, or process.
5. These incremental cycles are typically two to four weeks in duration and continue until the product is……...

Notes

4.2 Prototyping Model

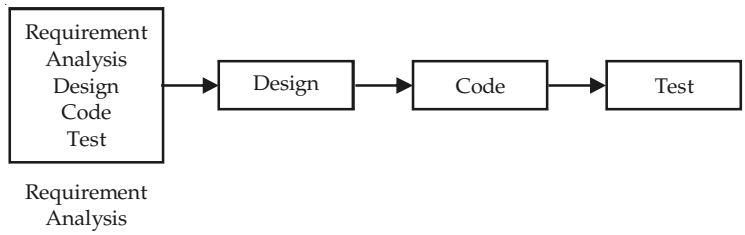
Since waterfall model shows some constraints; we study prototyping to counter these problems. The goal of a prototyping-based development process is to counter the first two limitations of the waterfall model. The basic idea here is that instead of freezing the requirements before any design or coding can proceed, a throwaway prototype is built to help understand the requirements. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding, and testing, but each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an actual feel of the system, because the interactions with the prototype can enable the client to better understand the requirements of the desired system. This results in more stable requirements that change less frequently.

Did u know? **What is the basic idea of prototyping model?**

The basic idea here is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements. In such situations, letting the client "play" with the prototype provides invaluable and intangible inputs that help determine the requirements for the system. It is also an effective method of demonstrating the feasibility of a certain approach. This might be needed for novel systems, where it is not clear that constraints can be met or that algorithms can be developed to implement the requirements. In both situations, the risks associated with the projects are being reduced through the use of prototyping. The process model of the prototyping approach is shown in Figure 4.2.

Figure 4.2: The Prototyping Model



A development process using throwaway prototyping proceeds as follows. The development of the prototype starts when the preliminary version of the requirements specification document has been developed. At this stage, there is a reasonable understanding of the system and its needs and of which needs are unclear or likely to change. After the prototype has been developed, the end users and clients are given an opportunity to use the prototype. Based on their experience, they provide feedback to the developers regarding the prototype: what is correct, what needs to be modified, what is missing, what is not needed, etc. Based on the feedback, the prototype is modified to incorporate some of the suggested changes that can be done easily, and then the users and the clients are again allowed to use the system. This cycle repeats until, in the judgment of the prototypers and analysts, the benefit from further changing the system and obtaining feedback is outweighed by the cost and time involved in making the changes and obtaining the feedback. Based on the feedback, the initial requirements are modified to produce the final requirements specification, which is then used to develop the production quality system.

Notes

In prototyping, for the purposes of requirement analysis to be feasible, its cost must be kept low. Consequently, only those features are included in the prototype that will have a valuable return from the user experience. Exception handling, recovery, and conformance to some standards and formats are typically not included in prototypes. In prototyping, as the prototype is to be discarded, there is no point in implementing those parts of the requirements that are already well understood. Hence, the focus of the development is to include those features that are not properly understood. Because the prototype is to be thrown away, only minimal documentation needs to be produced during prototyping. For example, design documents, a test plan, and a test case specification are not needed during the development of the prototype. Another important cost-cutting measure is to reduce testing. Because testing consumes a major part of development expenditure during regular software development, this has a considerable impact in reducing costs. By using these types of cost-cutting methods, it is possible to keep the cost of the prototype low.

Prototyping is often not used, as it is feared that development costs may become large. However, in some situations, the cost of software development without prototyping may be more than the prototyping. There are two major reasons for this. First, the experience of developing the prototype might reduce the cost of the later phases when the actual software development is done. Secondly, in many projects, the requirements are constantly changing, particularly when development takes a long time. We saw earlier that changes in requirements at a later stage, during development, substantially increase the cost of the project. By elongating the requirements analysis phase (prototype development does take time), the requirements are "frozen" at a later time, by that time they are likely to be more developed and, consequently, more stable. In addition, because the client and use to get experience with the system, it is more likely that the requirements specified after the prototype will be closer to the actual requirements. This again will lead to fewer changes in the requirements at a later time. Hence, the costs incurred due to changes in the requirements may be substantially reduced by prototyping. Hence, the cost of the development after the prototype can be substantially less than the cost without prototyping; we have already seen how the cost of developing the prototype itself can be reduced.

Task In a group of four analyze this statement: "The basic reason for little common use of prototyping is the cost involved in this built-it-twice approach."

Prototyping is catching on. It is well suited for projects where requirements are hard to determine and the confidence in obtained requirements is low. In such projects, a waterfall model will have to freeze the requirements in order for the development to continue, even when the requirements are not stable. This leads to requirement changes and associated rework while the development is going on. Requirements frozen after experience with the prototype are likely to be more stable. Overall, in projects where requirements are not properly understood in the beginning, using the prototyping process model can be the most effective method for developing the software. It is an excellent technique for reducing some types of risks associated with a project. We will further discuss prototyping when we discuss requirements specification and risk management.

Advantages of Prototyping

- Users are actively involved in the development
- It provides a better system to users, as users have natural tendency to change their mind in specifying requirements and this method of developing systems supports this user tendency.

Notes

- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier as the system is mode side by side.
- Quicker user feedback is available leading to better solutions.

Disadvantages of Prototyping

- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.

Self Assessment

Fill in the blanks:

6. Development of the …….. obviously undergoes design, coding, and testing, but each of these phases is not done very formally or thoroughly.
7. Prototyping is an attractive idea for complicated and large systems for which there is no …….. process or existing system to help determine the requirements.
8. After the prototype has been developed, the end users and ……. are given an opportunity to use the prototype.
9. Prototyping is often not used, as it is feared that development costs may become…….
10. Requirements frozen after experience with the prototype are likely to be more……..

4.3 The Spiral Model

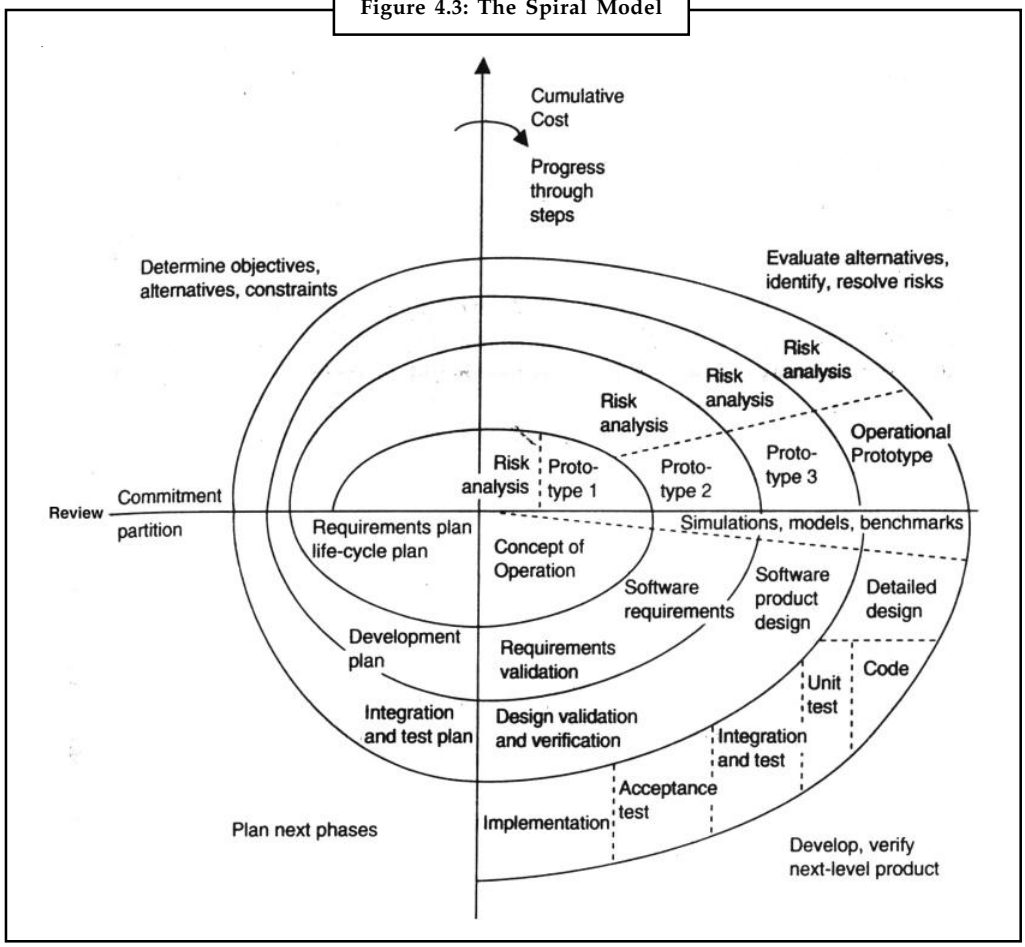
As it is clear from the name, the activities in this model can be organized like a spiral that has many cycles. The radial dimension represents the cumulative cost incurred in accomplishing the steps done so far, and the angular dimension represents the progress made in completing each cycle of the spiral. The model is shown in Figure 4.3.

Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exist. This is the first quadrant of the cycle (upper-left quadrant). The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project. Risks reflect the chances that some of the objectives of the project may not be met. The next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping. Next, the software is developed, keeping in mind the risks. Finally, the next stage is planned.

The development step depends on the remaining risks. For example, if performance or user-interface risks are considered more important than the program development risks, the next step may be an evolutionary development that involves developing a more detailed prototype for resolving the risks. On the other hand, if the program development risks dominate and the previous prototypes have resolved all the user-interface and performance risks, the next step will follow the basic waterfall approach.

Notes

Figure 4.3: The Spiral Model



The risk-driven nature of the spiral model allows it to accommodate any mixture of a specification-oriented, prototype-oriented, simulation-oriented, or some other type of approach.

Notes An important feature of the model is that each cycle of the spiral is completed by a review that covers all the products developed during that cycle, including plans for the next cycle.

The spiral model works for development as well as enhancement projects. In a typical application of the spiral model, one might start with an extra round zero, in which the feasibility of the basic project objectives is studied. These project objectives may or may not lead to a development/ enhancement project. Such high-level objectives include increasing the efficiency of code generation of a compiler, producing a new full-screen text editor and developing an environment for improving productivity. The alternatives considered in this round are also typically very high-level, such as whether the organization should go for in-house development, or contract it out, or buy an existing product. In round one, a concept of operation might be developed. The objectives are stated more precisely and quantitatively and the cost and other constraints are defined precisely. The risks here are typically whether or not the goals can be met within the constraints. The plan for the next phase will be developed, which will involve defining separate activities for the project. In round two, the top-level requirements are developed. In succeeding rounds, the actual development may be done.

Notes

This is a relatively new model; it can encompass different development strategies. In addition to the development activities, it incorporates some of the management and planning activities into the model. For high-risk projects, this might be a preferred model.

Advantages

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

Self Assessment

Fill in the blanks:

11. The spiral model is similar to the ..... model, with more emphases placed on risk analysis.
12. The spiral model has four phases: Planning, Risk Analysis, Engineering and.....
13. The baseline spiral, starting in the planning phase, requirements is gathered and ..... is assessed.

4.4 Concurrent Development Model

The concurrent development model, sometimes called concurrent engineering. The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states. For example, the engineering activity defined for the spiral model is accomplished by invoking the following tasks: prototyping and/or analysis modeling, requirements specification, and design.

The activity-analysis-may be in any one of the states noted at any given time. Similarly, other activities (e.g., design or customer communication) can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example, early in a project the customer communication activity has completed its first iteration and exists in the awaiting changes state. The analysis activity (which existed in the none state while initial customer communication was completed) now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the analysis activity moves from the under development state into the awaiting changes state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities.

Notes

4.5 A Final Comment on Evolutionary Models

This process model recognizes that software evolves over a period of time. It enables the development of an increasingly more complicated version of the software. The approach is iterative in nature. Specific evolutionary process models are

1. Prototyping Model
2. Spiral Model
3. Current development model

Self Assessment

Fill in the blanks:

14. The …….. may be in any one of the states noted at any given time.
15. All activities exist concurrently but reside in different……..

Caselet

Manufacturing hub called India

With the German engineering technology major, Siemens, planning to make India the global hub for manufacturing its key steel plant equipment, it joins the growing ranks of firms eyeing the country as a launching pad for supplies to Asian markets. India has the manufacturing and engineering capabilities; it has a pool of skilled expertise, and its size offers it a strategic advantage for servicing markets from Myanmar down to Australia and West Asia, if necessary. The Asia-Pacific region is the epicentre of economic expansion at the moment.

Slowly but steadily, almost unnoticed, a transformation of long-term significance is taking place in the economy. Indian manufacturing is diversifying not as a result of any policy initiative but because of conditions on the ground that global players are using to their advantage.

Anecdotal evidence suggests that India may well be on its way to becoming a global manufacturing hub. The pace seems to have increased at a time when the Western economies have yet to witness a pick-up in investments in their own economies.

A couple of years ago, Capgemini Consulting Services undertook a survey of 340 from among the Fortune 5000 global manufacturing companies and, in its report, observed that India could well overtake China as a global manufacturing hub. Most of the respondents stressed that India was large on their radar screen for outsourcing manufacturing over the next three to four years.

The Capgemini report was issued in 2007; since then, a host of firms from Japanese automakers to telecom equipment manufacturers and, now, Siemens are packing their bags for India. The country, the authors of the report were quoted in press reports, could well witness a change, with manufacturing overtaking IT as the driver of growth.

Contd...

Notes

Valorising Services

That change is not yet evident. India's economic successes are still identified with the IT sector and, more specifically, with a handful of firms that dominate the space. Services contribute the most to the GDP; software and business process outsourcing represent the brightest face of exports — icons of the country's passage into post-modernity.

The services sector dominates the economy in more ways than one; in terms of employment preferences, BFSI (Banking, Financial Services and Insurance) has overtaken manufacturing to a considerable degree. Most management or IIT graduates would prefer BFSI or IT firms for jobs over manufacturing even though salaries in the former are more prone to the vicissitudes of the business cycle.

In 2009, placement salary offerings in IT and BFSI dropped 25 per cent over 2007-08, while salaries in the manufacturing sector fell far less. Yet, the BFSI sector grabbed 39 per cent of graduates from the Indian Institutes of Management (IIMs) followed by consulting firms that attracted 24 per cent.

Manufacturing together with the media and rating agencies had to settle for a mere 6 per cent of the future cream of Indian managerial talent.

It is this star-lit sheen of IT and BFSI sectors that also blinds analysts, keen to advice the rest of the world just how it should go about getting prosperous.

India's liberalisation has focused largely on the financial and IT sectors, and the fact that both are in their own ways connected to the global economy offers the impression of the country participating in the creation of global wealth.

In their paper titled, "Service with a smile: A New growth Engine for poor countries" published on the Web site www.voxeu.org, Ejaz Ghani, Arti Grover (from The World Bank) and Homi Kharas from the Brookings Institution argue that services provide poor countries the easiest and fastest route to prosperity. At one time the privilege of rich countries, the globalisation of services has created new opportunities for countries mired in poverty. That data can substantiate their claim as accomplished fact does not detract from the strong apprehension and deep scepticism that must accompany this strategy for growth.

Chinks in the Model

The world knows what happens to countries that decide to become rich quick through services-led economic growth. Iceland is a case in point. A traditional cod-fishing based economy with a high level of stability, it could not resist the temptation that global finance threw its way to become a superstar, the crowning example of financial services-inspired prosperity, only to come to grief when Lehmann Brothers collapsed. The morality tale is yet to conclude, with Portugal the latest victim of the globalisation of services.

On the other hand, countries that have a strong manufacturing base, such as the emerging economies and India and China could, with the help of efficient and prescient monetary and exchange controls, contain the contagion unleashed by Wall Street on the world.

The fact that 'services' have become globalised distorts the differences between services exported by the US and the EU members such as France and Germany (finance capital and universal banking) on the one hand, and those adopted by poor countries, from tourism to business process outsourcing and call centres on the other; the fact of 'success' obfuscates the inherent inequalities.

Contd...

Notes

Manufacturing outsourcing, too, contains its systems of domination with its jealous protection of advanced technologies.

But the advantage of technology spill-over and its immersion into the local economy can hardly be denied. China is a shining example of the intuitive transfer of technologies that creates a more sustainable and inclusive economy than one propelled by call centres or tourism.

4.6 Summary

- One of the major problems here is the way of dealing with the development of software.
- The prototypes produced are being used in the next phase. Because many prototypes are produced that are redefined further and further, the whole software might become a patchwork of features.
- Hence the whole approach might only work for small projects.
- Since waterfall model shows some constraints; we study prototyping to counter these problems.
- The basic idea of prototyping is that instead of freezing the requirements before any design or coding can proceed, a throwaway prototype is built to help understand the requirements.
- Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase.
- Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.
- In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.

4.7 Keywords

EVO: Evolutionary Development Model

Prototype Model: The goal of a prototyping-based development process is to counter the first two limitations of the waterfall model.

Spiral Model: In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.

4.8 Review Questions

1. The evolutionary model could be seen as one of the classic iterative activity based models. Explain.
2. Scrutinize what are the advantages and disadvantages of evolutionary model.
3. Evolutionary model allows the marketing department access to early deliveries, facilitating development of documentation and demonstrations. Give reasons and examples to support your answers.
4. Compare and contrast evolutionary and concurrent development model. Give examples.

Notes

5. The goal of prototyping based development is to counter the first two limitations of the waterfall model. Discuss.
6. Development of the prototype obviously undergoes design, coding and testing. Justify your answers with examples.
7. The client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system. Discuss.
8. As it is clear from the name, the activities in this model can be organized like a spiral that has many cycles. Explain.
9. The radial dimension represents the cumulative cost. Substantiate with suitable examples.
10. Why concurrent development model, sometimes called concurrent engineering?

Answers: Self Assessment

1.	spiral	2.	reduction
3.	co-operation	4.	product
5.	shipped	6.	prototype
7.	manual	8.	clients
9.	large	10.	stable
11.	incremental	12.	Evaluation
13.	Risk	14.	activity-analysis
15.	states		

4.9 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering*, 2nd Edition, PHI.

Richard Fairpy, *Software Engineering Concepts,* Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering,* 6th Edition, Pearson Education

Online links

http://codebetter.com/raymondlewallen/2005/07/13/software-development-life-cycle-models/

http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/spiral.pdf

Notes

Unit 5: An Agile View of Process

CONTENTS

Objectives
Introduction
5.1 An Agile View of Process
5.1.1 What is Agility?
5.2 Agile Process Model
5.2.1 XP (Extreme Programming)
5.2.2 Adaptive Software Development
5.2.3 DSDM
5.2.4 Scrum
5.2.5 Crystal
5.2.6 Feature-Driven Development (FDD)
5.2.7 AM
5.3 Summary
5.4 Keywords
5.5 Review Questions
5.6 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize the agility
- Describe Agile Process Models

Introduction

Agile Modeling is a practice based software process whose scope is to cover both architectural and software process point of view. An agile philosophy for software engineering stresses four key issues: the importance of self-organizing teams that have control over the work they perform: communication and collaboration between team members and between practitioners and their customers, a recognition that change represents an opportunity; and an emphasis on rapid delivery of software that satisfies the customer. Agile process models have been designed to address each of these issues.

5.1 An Agile View of Process

In the software industry, a large number of projects fail and billions of dollars are spent on failed software projects. Lack of an end user involvement, poor requirements, and unrealistic schedules are some of the top reasons of such failure. Agile software development is an approach that addresses these problems through a real communication between programmers and customers. Thus, there is a need to quantify software agility. In this topic, an approach for quantifying software agility is provided by modeling the key concepts in agile software and proposing a

measure that can be used in representing how agile a software development process is. The proposed measure employs information entropy as the main concept related to software agility.

5.1.1 What is Agility?

To become Agile, most organizations will need to change their perspective. Most of our assumptions about business, about technology and organizations are at least 50 years old. They have outlived their time. As a result, we are preaching, teaching, and practicing policies that are increasingly at odds with reality and therefore counterproductive.

Agility isn't a one-shot deal that can be checked off the organizational initiative list. Agility is a way of life, a constantly emerging and changing response to business turbulence. Critics may counter, "Agility is merely waiting for bad things to happen, then responding. It is a fancy name for lack of planning and ad hoc-ism." But Agile organizations still plan; they just understand the limits of planning.

Notes Although the defining issue of agility involves creating and responding to change, there are three other components that help define agility: nimbleness and improvisation, conformance to actual, and balancing flexibility and structure.

Creating and Responding to Change

Is agility merely a matter of reacting to stimuli, like an amoeba, or is it something more? My preferred definition of agility has two aspects:

Agility is the ability to both create and respond to change in order to profit in a turbulent business environment.

Agility is not merely reaction, but also action. First and foremost, Agile organizations create change, change that causes intense pressure on competitors. Creating change requires innovation, the ability to create new knowledge that provides business value. Second, Agile organizations have an ability to react, to respond quickly and effectively to both anticipated and unanticipated changes in the business environment.

Agility means being responsive or flexible within a defined context. Part of the innovative, or proactive, piece involves creating the right context—an adaptable business organization or an adaptable technology architecture, for example. Innovation is understanding that defined context and anticipating when it needs to be altered. The problem with many traditional software development and project management approaches is that they have too narrowly defined the context; that is, they've planned projects to a great level of task detail, leaving very little room for agility to actually happen.

"We think Agile is both being able to respond quickly (which means you have to know that you have to respond quickly—not an easy thing to do) as well as being able to anticipate when the rules or principles are changing or, more importantly, can be changed," says Bob. "Organizations that do this are often called "lucky"—in the right place at the right time. Some undoubtedly are, while a few have created the necessary infrastructure to exploit the change." Part of the cost of agility is mistakes, which are caused by having to make decisions in an environment of uncertainty. If we could wait to make product development decisions, they might be better ones; however, the delay could well obviate the need for the decision at all, because aggressive competitors may get their product to market while we dither. "This fact leads to a sort of organizational uncertainty principle: The faster your decision-making cycle, the less assurance you can have that you're making the best possible decision," says David Freedman (2000).

Notes

Task Agility is not merely reaction, but also action. Analyze.

Nimbleness and Improvisation

In our volatile economy, companies need to enhance their "exploration" skills at every level of the organization. Good explorers are Agile explorers—they know how to juggle and improvise. Indiana Jones was a good explorer, somehow living through every outlandish adventure. Agility means quickness, lightness, and nimbleness—the ability to act rapidly, the ability to do the minimum necessary to get a job done, and the ability to adapt to changing conditions. Agility also requires innovation and creativity—the ability to envision new products and new ways of doing business. In particular, IT organizations have not done an adequate job of balancing the needs of exploration and optimization.

The actors in the movie Crouching Tiger, Hidden Dragon display incredible agility—running lightly along the tiniest tree branches and showing extraordinary dexterity in sword fighting. Beginning martial arts students are clumsy, not Agile. They become skilled, and Agile, from long hours of training and effective mentoring. Sometimes their drills are repetitive and prescriptive, but only as part of learning.

Agility also requires discipline and skill. A skilled software designer can be more Agile than a beginner because he or she has a better sense of quality. Beginners, with little sense of what is good and what is not, can just revolve in circles—lots of change but not much progress.

"We view the issue as one of almost 'disciplined messiness,'" remarked Bob in an email exchange. "You need to have a very good discipline in place to be able to respond in turbulent times, yet simultaneously know when to be 'undisciplined.' I view anticipation to be actively seeking situations where the generally accepted guiding rules or principles no longer apply, or where shortcuts are the least risky approach to take to gaining some objective. To be able to understand when the rules don't apply, you need to completely understand when they do."

Example: Picasso had to become an accomplished fine arts painter to get to a point where he was able to move past that perception of "good art" and create abstract painting. He had to be skilled before he could be Agile.

Agile individuals can improvise, they know the rules and boundaries, but they also know when the problem at hand has moved into uncharted areas. They know how to extend their knowledge into unforeseen realms, to experiment, and to learn. When critical things need to get done, call on the great improvisers.

Improvisation makes great jazz bands. From a few key structural rules, jazz bands improvise extensively. Having a solid foundation enables their tremendous flexibility without allowing the music to degenerate into chaos. The proponents of business process reengineering and software engineering methodologies probably blanch at the thought that improvisation, rather than carefully articulated processes, are key to success. Yet in today's turbulent environment, staff members with good balancing, judging, and improvisational skills are truly invaluable.

Conformance to Actual

There are two answers to this constant question. First, control focuses on boundaries and simple rules rather than prescriptive, detailed procedures and processes. The second aspect of control, though simple in concept, is completely foreign to many managers.

Notes

Agile projects are not controlled by conformance to plan but by conformance to business value.

"The major problem with planning," say Shona Brown and Kathleen Eisenhardt (1998), "is that plans are virtually always wrong." If we accept the notion of constant change and turbulence, then plans are still useful as guides, but not as control mechanisms. In high-change environments, plans—and the traditional contracts that are derived from plans—are worse than useless as control mechanisms because they tend to punish correct actions. If we deliver a working software product, with features our customers accept, at a high level of quality, within acceptable cost constraints, during the specified time-box, then we have delivered business value. Developers don't get to say, "This is valuable." Customers do. Every three to six weeks, customers tell developers that acceptable business value has been delivered—or not. If it hasn't, the project is cancelled or corrective action is taken. If it has, the project continues for another iterative cycle.

We may look at the plan and say, "Well, because we learned this and this and this during the iteration, we only got 23 of the 28 planned features done." That is useful information for planning our next iteration, but not for control. When we started the project three months ago, we only planned 18 features for this last cycle, and half the features we did deliver were different than the ones in the original plan. We accept that talented people, who are internally motivated, who must work in a volatile environment, who understand the product vision, will do the best they can do. If they don't conform to the plan, the plan was wrong. If they delivered business value, then whether the plan was "right" or not is immaterial.

Caution If they conformed to the plan and the customers aren't happy with the delivered business value, it doesn't matter that they conformed to the plan.

An entire generation of project managers has been taught, by leading project management authorities, to succeed at project management by conforming to carefully laid-out, detailed task plans. Conformance to plan means locking ourselves into a outdated, often irrelevant plan that some project manager cooked up in great haste (which, of course, precluded talking to developers) 18 months ago when the world was different. Conformance to plan may get a project manager brownie points with the tightly wound project management authorities, but it won't deliver business value in volatile, high-speed environments.

An exploration perspective contrasts with that of optimization. Take, for example, the use of schedule deadlines. While a schedule may appear to be a schedule, each perspective utilizes dates as a control mechanism in different ways. Optimization cultures use dates to predict and control—they view schedule as achievable, along with the other objectives, and see deviations from the plan as poor performance. Exploration cultures, however, view dates much differently. They basically see dates as a vehicle for managing uncertainty and thereby helping to make necessary scope, schedule, and cost tradeoffs. Exploration cultures, to be sure, use dates as performance targets, but the primary focus is bounding the uncertainty, not predicting dates.

Balancing Flexibility and Structure

It would be very easy to either "plan everything" or "plan nothing," but the really interesting question is how to juggle the two, how to define the context narrowly enough to get something meaningful done, but not so narrowly that we fail to learn and adapt as we go along. The fundamental issue remains one's primary perspective: to anticipate or to depend on resilience and responsiveness. Aaron Wildavsky, a social scientist, writes about this issue and offers an example of the difference between Silicon Valley and Boston. Basically, he believes, the reason Silicon Valley is the primary technology center of the world is that it depends on its resilience to be able to respond to rapid change, whereas the Boston crowd leans more heavily toward

Notes

anticipation (Postrel 1998). Does that mean no one in Boston ever responds to change and no one in Silicon Valley ever plans? It's not so much either/or as it is a fundamental, underlying philosophy within which situations are juggled accordingly.

Being Agile means trusting in one's ability to respond more than trusting in one's ability to plan. Good explorers both anticipate when the rules of the game have changed (or are about to change)—that is, they define the context—and can also operate flexibly within a given rule set. Obviously, if the rule set itself is too prescriptive, it leaves no room for agility. Without some rule set, however, agility can become rudderless reaction.

Self Assessment

Fill in the blanks:

1. Agility isn't a one-shot deal that can be checked off the organizational …….. list.
2. Agility is a way of life, a constantly emerging and changing response to …… turbulence.
3. Agile projects are not controlled by ……. to plan but by conformance to business value.

5.2 Agile Process Model

Computer science is a young science. Computer programmers my age were trained by engineers. That training dictated how we approached software development for an entire generation. But now after decades of building software to be expensive, unwanted, and unreliable we have come to realize software is different. Building software is more like creating a work of art, it requires creativity in design and ample craftsmanship to complete. Software remains malleable, often illogical, and incomplete forever. Agile software development is based on fundamental changes to what we considered essential to software development ten years ago.

The most important thing to know about Agile methods or processes is that there is no such thing. There are only Agile teams. The processes we describe as Agile are environments for a team to learn how to be Agile.

We realize the way a team works together is far more important than any process. While a new process can easily improve team productivity by a fraction, enabling your team to work effectively as a cohesive unit can improve productivity by several times. Of course to be eligible for such a big improvement you must be working at a fraction of your potential now. Unfortunately, it isn't that uncommon.

The most brilliant programmers alive working competitively in an ego-rich environment can't get as much done as ordinary programmers working cooperatively as a self disciplined and self-organizing team. You need a process where team empowerment and collaboration thrive to reach your full potential.

The second change is making the customer, the one who funds the software development, a valuable and essential team member. When the dead line gets close a traditional approach to reducing scope is to let the developers decide what will work properly and what won't. Instead let the customer make scope decisions a little at a time throughout the project.

When your customer, or domain expert works directly with the development team everyone learns something new about the problem. True domain expertise and experience is essential to finding a simple, elegant, correct solution. A document can have plenty of information, but real

Notes

knowledge is hard to put on paper. Left alone programmers must assume they know everything they need. When asking questions is difficult or slow the knowledge gap grows. The system will get built, but it won't solve the problem like one guided by an expert on a daily basis.

Perhaps the biggest problem with software development is changing requirements. Agile processes accept the reality of change versus the hunt for complete, rigid specifications. There are domains where requirements can't change, but most projects have changing requirements. For most projects readily accepting changes can actually cost less than ensuring requirements will never change.

We can produce working software starting with the first week of development so why not show it to the customer? We can learn so much more about the project requirements in the context of a working system. The changes we get this way are usually the most important to implement. Agile also means a fundamental change in how we manage our projects. If working software is what you will deliver then measure your progress by how much you have right now. We will change our management style to be based on getting working software done a little at a time. The documents we used to create as project milestones may still be useful, just not as a measure of progress.

Notes Instead of managing our activities and waiting till the project ends for software, we will manage our requirements and demonstrate each new version to the customer. It is a hard change to make but it opens up new ways to develop software.

"Agile process model" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations or parts and do not directly involve long term planning. The project scope and requirements are clearly laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.

Each iteration is considered as a short time "frame" in Agile process model which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding and testing before a working product is demonstrated to the client.

Agile process model can be implemented with the help of various methods such as Extreme Programming, Essential Unified Process, DSDM, Agile Data Method, Agile Modeling, Agile Unified Process and so on. Most models promote a business approach that maps software development with customer needs and company goals.

"Flash Application Development" is a professional IT solutions company committed to deliver higher-end IT solutions based on Agile process model. Agile methods help minimize overall risk and lets the project adapt to changes quickly, which is very critical if business has to succeed in today's competitive world.

5.2.1 XP (Extreme Programming)

The first Extreme Programming project was started March 6, 1996. Extreme Programming is one of several popular Agile Processes. It has already been proven to be very successful at many companies of all different sizes and industries world wide.

Notes

Extreme Programming is successful because it stresses customer satisfaction. Instead of delivering everything you could possibly want on some date far in the future this process delivers the software you need as you need it. Extreme Programming empowers your developers to confidently respond to changing customer requirements, even late in the life cycle.

Extreme Programming emphasizes teamwork. Managers, customers, and developers are all equal partners in a collaborative team. Extreme Programming implements a simple, yet effective environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.

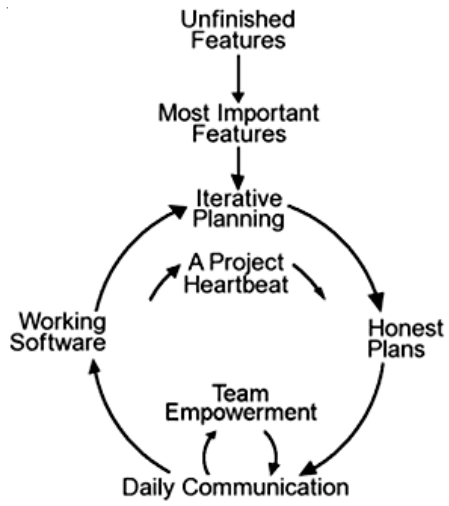
Extreme Programming improves a software project in five essential ways; communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. Every small success deepens their respect for the unique contributions of each and every team member. With this foundation Extreme Programmers are able to courageously respond to changing requirements and technology.

The most surprising aspect of Extreme Programming is its simple rules. Extreme Programming is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen. The rules may seem awkward and perhaps even naive at first, but are based on sound values and principles.

Our rules set expectations between team members but are not the end goal themselves. You will come to realize these rules define an environment that promotes team collaboration and empowerment, that is your goal. Once achieved productive teamwork will continue even as rules are changed to fit your company's specific needs.

This flow chart shows how Extreme Programming's rules work together. Customers enjoy being partners in the software process, developers actively contribute regardless of experience level, and managers concentrate on communication and relationships. Unproductive activities have been trimmed to reduce costs and frustration of everyone involved.

Figure 5.1 Extreme Programming



Notes

Task Visit the website of which incorporates the extreme programming and make a note on the advantages of extreme programming.

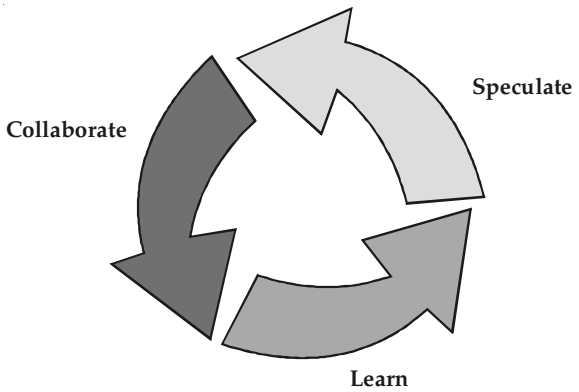
5.2.2 Adaptive Software Development

Speculate

In complex environments, planning is a paradox. According to CAS theory, outcomes are unpredictable. Yet wandering around, endlessly experimenting on what a product should look like is not likely to lead to profitability either. "Planning," whether it is applied to overall product specifications or detail project management tasks, is too deterministic a word. It carries too much historical baggage. "Speculate" is offered as a replacement.

When we speculate, it's not that we don't define a mission to the best of our ability. (I use "mission" as a summary term for objectives, vision, goals, and outline requirements.) It simply admits the obvious — in some important dimensions of our mission statements, we are more than likely wrong. Whether we misread our customer's needs, or technology changes, or competitors come out with a better mousetrap, the probability of mistakes is high. So let's be honest, postulate a general idea of where we are going, and put mechanisms in place to adapt. In a complex environment, following a plan produces the product you intended — just not the product you need.

Figure 5.2: The Adaptive Cycle



Collaborate

Managing in a complex environment is scary as hell — it is also a blast. If we can't predict (plan), then we can't control in the traditional management sense. If we can't control, then a significant set of current management practices is no longer operable, or more specifically, only operable for those parts of the development process that are predictable.

Did u know? **What are the uses of collaboration in adaptive software development?**

Collaboration, in this context, portrays a balance between managing the doing (the main thrust of traditional management) and creating and maintaining the collaborative environment needed for emergence.

Notes

As projects become increasingly complex, the balance swings much more toward the latter. At times it seems almost mystical, but in field after field, from physics to cellular automata to some of my client's projects, emergence has, well, . . . emerged . We have all experienced emergent results on some special project, but it somehow seemed nearly accidental, not something to count on in a crunch. CAS provides some comfort that it is not accidental.

For a project manager, maintaining this balance means two things. First, he or she has to decide which parts of the project are predictable. For example, we can predict that without appropriate configuration control procedures, a software project of any size can implode. For parts that are unpredictable, he or she has to establish an environment in which the wild and wonderful properties of emergence — basically open, collaborative, messy, exciting, diverse, anxiety-ridden, and emotion-laden — can exist.

Unfortunately, at least for some people, CAS postulates certain conditions for emergent behavior. The most important is that it happens at the edge of chaos. Whether in physics, biology, business, or human behavior, it appears that there are three broad categories of environments — stable, unstable (or chaotic), and a transition zone labeled the "edge of chaos."

The edge of chaos is the constantly shifting battle zone between stagnation and anarchy, the one place where a complex system can be spontaneous, adaptive, and alive.

In human terms, being in chaos is analogous to being psychotic. So the trick is to lead a project team away from the familiar and the stable toward chaos, but not all the way. Success comes to those who can hold anxiety, who can attune themselves to paradox and uncertainty. Innovation, creativity, and emergent results are born in the transition zone at the edge of chaos.

Learn

Collaborative activities build products. Learning activities expose those products to a variety of stakeholders to ascertain value. Customer focus groups, technical reviews, beta testing, and postmortems are all practices that expose results to scrutiny.

Learning, according to the dictionary, is gaining mastery through experience. In an adaptive environment, learning challenges all stakeholders, including both developers and customers, to examine their assumptions and use the results of each development cycle to learn the direction of the next. The cycles need to be short, so teams can learn from small rather than large mistakes. They also need to be double-loop, so teams learn both about product changes and more fundamental changes in underlying assumptions about how the products are being developed.

Speculate—Collaborate—Learn

If you examine the speculate — collaborate — learn cycle, even briefly, it becomes obvious that the three stages overlap. It is difficult to collaborate without learning or to learn without collaborating. They are purposely messy, nonlinear, overlapping terms — how could terms that describe an adaptive framework be otherwise?

For many project leaders and project teams, adaptive development is a terrifying prospect. First, we knock away the foundation pillar of cause and effect, so we can't say for sure what needs to be done next. Next we force the team into meeting deliverable goals, but we admit we don't know exactly what they are. Then when they then get anxious and concerned about all the "seemingly inefficient" groping around for solutions (because of the needed diversity and multiple interactions), we have to say that high anxiety is part of the new game, and it won't go away. And finally, when a successful product emerges, to many it seems almost accidental. It is not a place for the timid.

Notes

Are there organizations practicing adaptive development? I would offer Microsoft as an example of a company that excels because its management style exemplifies adaptive development.

Notes For all the techniques and practices discussed in books like Microsoft Secrets, we believe Microsoft prospers because at the core of these practices lies the abandonment of determinism and an embrace of the messiness of speculation — collaboration — learning.

Through the Lens of Adaptive Software Development

The lens of ASD offers a different perspective on software management practices. The following is a brief examination of two of these, quality and RAD, both of which have ramifications for gathering requirements.

Do it Wrong the First Time

Using our new lens, let us look at the current state of software quality management practices epitomized by the phrase "Do it right the first time." In a complex environment, "Do it right the first time" is a recipe for failure.

First, how can we predict what right means? In the early stages, if the delivery time horizon isn't too far out, we may be able to speculate on what the generally correct direction is, but defining "right" borders on fantasy. Even if we could define right, doing it the first time makes no sense for other than trivial products. The first time assumes we understand the cause and effect, the specific algorithm of getting to the final product from our initial starting position, and the needs of all stakeholders — it says we know it all.

RAD Practices: A Step in the Right Direction

RAD practices generally involve some combination of the following:

- Evolutionary life cycle
- Customer focus groups, JAD sessions, technical reviews
- Timeboxed project management
- Continuous software engineering
- Dedicated teams with war rooms

Most RAD projects I've been associated with over the last five years have had an adaptive, emergent flavor. More recently, as my practice has focused on speculating and establishing collaborative environments, emergent characteristics have become even more pronounced.

RAD is an anathema to many IT organizations. It works, but many would prefer it did not. Their defense is to label RAD as "hacking" and/or to relegate it to small (and by implication relatively unimportant) development projects. But Microsoft and others have produced incredibly large and complex software using techniques comparable to RAD. RAD is scary to traditional organizations because it raises questions about their fundamental world view.

Example: RAD practices and the Microsoft process are both examples of adaptive development in action. Giving them a label (i.e., adaptive development) and realizing there is a growing body of scientific knowledge (i.e., CAS theory) that begins to help explain why they work should provide a basis for more extensive use of these practices.

Notes

5.2.3 DSDM

Dynamic Systems Development Method (DSDM) is a structure based initially around Rapid Application Development (RAD), carried by its continuous user involvement in an iterative development and incremental approach which is approachable to changing requirements, in order to expand a system that meets the business needs on time and on budget. It is one of a number of Agile methods for developing software and forms part of the Agile Alliance.

DSDM was developed in the United Kingdom in the 1990s by a group of vendors and experts in the field of Information System (IS) development, the DSDM Consortium, unite their best-practice experiences. The DSDM group is a non-profit and vendor independent association which owns and administers the framework. The first version was finished in January 1995 and published in February 1995. The current version in use at this point in time (April 2006) is Version 4.2: Framework for Business Centered Development released in May 2003.

As an expansion of rapid application development, DSDM focuses on Information Systems projects that are distinguished by fixed schedules and budgets. DSDM addresses the ordinary reasons for information systems project failure counting exceeding budgets, missing deadlines, and lack of user participation and top management commitment.

DSDM identifies that projects are limited by time and resources, and plans consequently to meet the business needs. In order to attain these goals, DSDM encourages the use of RAD with the ensuing danger that too many corners are cut. DSDM applies some principles, roles, and techniques.

Notes In some circumstances, there are possibilities to integrate practices from other methodologies, such as the Select Perspective, Extreme Programming (XP), and PRINCE2, as complements to DSDM. Another agile method that has some similarity in process and concept to DSDM is Scrum.

Principles of DSDM

There are 9 fundamental principles of DSDM consisting of four foundations and five starting-points for the structure of the method. These values form the keystone of development using DSDM.

- User participation is the main key in running a well-organized and effectual project, where both users and developers share a workplace, so that the decisions can be made precisely.
- The project team must be authorized to make decisions that are important to the progress of the project, without waiting for higher-level approval.
- DSDM focuses on recurrent delivery of products, with supposition that to deliver something "good enough" earlier is always better than to deliver everything "perfectly" in the end. By delivering product frequently from an early stage of the project, the product can be tested and reviewed where the test record and review document can be taken into account at the next iteration or phase.
- The main criteria for acceptance of deliverable in DSDM are on delivering a system that addresses the current business needs. It is not so much directed at delivering a perfect system addressing all possible business needs, but focuses its efforts on critical functionality.

Notes

- Growth is iterative and incremental, driven by users' feedback to converge on an effective business solution.
- All changes during the growth are reversible.
- The high level scope and requirements should be base-lined before the project starts.
- Testing is carried out all through the project life-cycle.
- Communication and co-operation among all project stakeholders is required to be efficient and effective.

Prerequisites for using DSDM

In order for DSDM to be an accomplishment, a number of fundamentals need to be realized. First, there needs to be interactivity between the project team, future end users and higher management. This addresses well known breakdown of IS development projects due to lack of top management motivation and/or user involvement.

The second significant prerequisite for DSDM projects is the decomposability of the project. The possibility of decomposition into smaller parts allows the iterative approach, and activities, that are hard to prioritize, often causes delays. And that is accurately the effect that DSDM was developed to avoid. Another group of projects for which DSDM is not well-suited are safety-critical ones. The extensive testing and validation found in these kinds of projects collide with DSDM goals of being on time and on budget. Finally, projects that aim at reusable mechanism might not be well-suited for development using DSDM, because the demands on perfection are too high and collide with the 80%/20% principle described earlier.

The Phases of DSDM

The DSDM structure consists of three sequential phases, namely the pre-project, project life-cycle and post-project phases. The project phase of DSDM is the most detailed of the three phases. The project life-cycle phase consists of 5 stages that form an iterative step-by-step approach in developing an IS. The three phases and corresponding stages are explained extensively in the subsequent sections. For each stage/phase, the most significant activities are addressed and the deliverables are mentioned.

Phase 1: The Pre-project

In the pre-project phase applicant projects are identified, project funding is realized and project commitment is ensured. Handling these issues at an early stage avoids problems at later stages of the project.

Phase 2: The Project Life-cycle

The process summary in the figure above shows the project life-cycle of this phase of DSDM. It depicts the 5 stages a project will have to go through to create an IS. The first two stages, the possibility Study and Business Study are sequential phases that complement to each other. After these phases have been completed, the system is developed iteratively and incrementally in the Functional Model Iteration, Design & Build Iteration and Implementation stages. The iterative and incremental nature of DSDM will be addressed further in a later section.

- Feasibility Study
- Business Study
- Functional Model Iteration

Notes

- Design and Build Iteration
- Implementation

Phase 3: Post-project

The post-project phase makes sure the system operating effectively and efficiently. This is realized by maintenance, enhancements and fixes according to DSDM principles. The preservation can be viewed as continuing development based on the iterative and incremental nature of DSDM. Instead of finishing the project in one cycle usually the project can return to the previous phases or stages so that the previous step and the deliverable products can be refined.

Core Techniques of DSDM

- Timeboxing
- MoSCoW
- Prototyping
- Testing
- Workshop
- Modelling
- Configuration Management

Task Explain the phases of DSDM in your words with example.

5.2.4 Scrum

Scrum is an agile approach to software development. Rather than a full process or method, it is a framework. So instead of providing entire, detailed descriptions of how everything is to be done on the project, much is left up to the software development team. This is done because the team will know best how to solve the problem they are presented. This is why, for instance, a sprint planning meeting is described in terms of the desired outcome (a commitment to set of features to be developed in the next sprint) instead of a set of Entry criteria, Task definitions, Validation criteria, and Exit criteria (ETVX) as would be provided in most methodologies. Scrum relies on a self-organizing, cross-functional team. The scrum team is self-organizing in that there is no overall team leader who decides which person will do which task or how a problem will be solved. Those are issues that are determined by the team as a whole. The team is cross-functional so that everybody essential to take a feature from idea to implementation is involved.

These agile development teams are supported by two exact individuals: a Scrum Master and a product owner. The Scrum Master can be thought of as a coach for the team, helping team members use the Scrum framework to execute at their highest level. The product owner represents the business, customers or users and guides the team toward building the right product.

Scrum projects make development in a series of sprints, which are time boxed iterations no more than a month long. At the start of a sprint, team members consign to delivering some number of features that were listed on the project's product backlog. At the end of the sprint, these features are *done*—they are coded, tested, and integrated into the evolving product or system. At the end of the sprint a sprint review is conducted during which the team demonstrates

the new functionality to the product owner and other interested stakeholders who provide feedback that could influence the next sprint.

Scrum Methodology

For many developers in the software organization, the agile methodology is nothing new. Most persons know that agile was a direct response to the leading project management paradigm, waterfall, and borrows many ethics from lean manufacturing. In 2001, as this new management pattern began to pick up momentum, agile was formalized when 17 pioneers of the agile methodology met at the Snowbird Ski Resort in Utah and issued the Agile Manifesto. Their policy is now considered the foundational text for agile practices and principles. Most extremely, the manifesto spelled out the philosophy behind agile, which places a new emphasis on communication and collaboration; functioning software; and the flexibility to adapt to emerging business realities.

But for all of the pace the Agile Manifesto made in revising a philosophical approach to software development, it didn't offer the concrete processes that development teams depend on when deadlines — and stakeholders — start applying pressure. As a consequence, when it comes to the nuts and bolts of running a team with agile every day, association turn to particular subsets of the agile methodology. These include Crystal Clear, Extreme Programming, Feature Driven Development, Dynamic Systems Development Method (DSDM), Scrum, and others.

What's Unique about Scrum?

Of all the agile methodologies, Scrum is exclusive because it introduced the concept of "empirical process control." That is, Scrum uses the real-world development of a project — not a best guess or uninformed forecast — to plan and schedule release. In Scrum, projects are separated into succinct work cadences, known as sprints, which are typically one week, two weeks, or three weeks in duration. At the end of each sprint, stakeholders and team members meet to assess the progress of a project and plan its next steps. This allows a project's direction to be adjusted or reoriented based on completed work, not speculation or predictions.

Philosophically, this emphasis on an ongoing assessment of completed work is largely responsible for its popularity with managers and developers alike. But what allows the Scrum methodology to really work is a set of roles, responsibilities, and meetings that never change.

Caution If Scrum's capacity for adoption and flexibility makes it an appealing option, the stability of its practices give teams something to lean on when development gets chaotic.

The Roles of Scrum

Scrum has three primary roles: Product Owner, ScrumMaster, and team member.

1. ***Product Owner:*** In Scrum, the Product Owner is dependable for communicating the vision of the product to the development team. He or she must also stand for the customer's interests through requirements and prioritization. Because the Product Owner has the most power of the three roles, it's also the role with the most responsibility. In other words, the Product Owner is the single individual who must face the music when a project goes awry.

   The nervousness between authority and responsibility means that it's hard for Product Owners to strike the right balance of involvement. Because Scrum values self-organization

Notes

among teams, a Product Owner must fight the urge to micro-manage. At the same time, Product Owners must be available to answer questions from the team.

2. ***ScrumMaster:*** The ScrumMaster acts as a link between the Product Owner and the team. The ScrumMaster does not manage the team. Instead, he or she works to remove any impediments that are obstructing the team from achieving its sprint goals. In short, this role helps the team remain creative and productive, while making sure its successes are visible to the Product Owner. The ScrumMaster also works to advise the Product Owner about how to maximize ROI for the team.

3. ***Team Member:*** In the Scrum methodology, the team is responsible for completing work. Preferably, teams consist of seven cross-functional members, plus or minus two individuals. For software projects, a typical team comprises a mix of software engineers, architects, programmers, analysts, QA experts, testers, and UI designers. Each sprint, the team is responsible for determining how it will achieve the work to be completed. This grants teams a great deal of autonomy, but, similar to the Product Owner's situation, that liberty is accompanied by a responsibility to meet the goals of the sprint.

5.2.5 Crystal

The Crystal method is one of the mainly lightweight, adaptable approaches to software development. Crystal is really comprised of a family of methodologies (Crystal Clear, Crystal Yellow, Crystal Orange, etc.) whose unique characteristics are driven by several factors such as team size, system criticality, and project priorities. This Crystal family addresses the realization that each project may require a slightly tailored set of policies, practices, and processes in order to meet the project's unique characteristics.

Several of the key tenets of Crystal comprise teamwork, communication, and simplicity, as well as reflection to frequently adjust and improve the process. Like other agile methodologies, Crystal promotes early, ordinary delivery of working software, high user involvement, adaptability, and the removal of bureaucracy or distractions.

5.2.6 Feature-Driven Development (FDD)

FDD was initially developed and articulated by Jeff De Luca, with contributions by M.A. Rajashima, Lim Bak Wee, Paul Szego, Jon Kern and Stephen Palmer. The first personification of FDD occurred as a result of collaboration between De Luca and OOD thought leader Peter Coad. FDD is a model-driven, short-iteration process. It begins with found an overall model shape. Then it carry on with a series of two-week "design by feature, build by feature" iterations. The features are small, "useful in the eyes of the client" results. FDD designs the rest of the development process around feature delivery using the following eight practices:

1. Domain Object Modeling
2. Developing by Feature
3. Component/Class Ownership
4. Feature Teams
5. Inspections
6. Configuration Management
7. Regular Builds
8. Visibility of progress and results

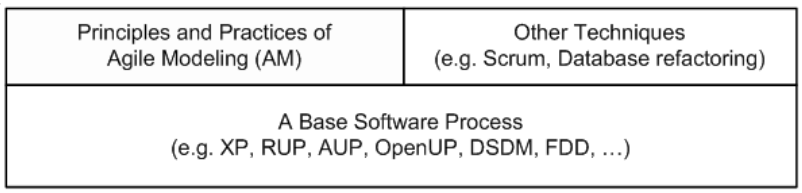
Notes

FDD suggests exact programmer practices such as "Regular Builds" and "Component/Class Ownership". FDD's proponents claim that it scales more uncomplicatedly than other approaches, and is better suited to larger teams. Unlike other agile approaches, FDD explains specific, very short phases of work which are to be accomplished separately per feature. These include Domain Walkthrough, Design, Design Inspection, Code, Code Inspection, and Promote to Build.

5.2.7 AM

Agile Modeling (AM) is a practice-based process for effectual modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a anthology of values, principles, and practices for modeling software that can be applied on a software development project in an effective and lightweight manner. As you see in Figure 5.3 AM is meant to be tailored into other, full-fledged methodologies such as XP or RUP, enabling you to develop a software process which truly meets your needs.

Figure 5.3: AM Enhances other Software Processes



The principles of AM, adopting and extending those of extreme Programming v1, are communication, simplicity, feedback, courage, and humility. The keys to modeling success are to have effectual communication between all project stakeholders, to strive to expand the simplest solution possible that meets all of your needs, to obtain feedback regarding your efforts often and early, to have the courage to make and stick to your decisions, and to have the humility to admit that you may not know everything, that others have value to add to your project efforts.

AM is based on a collection of principles, such as the significance of assuming unfussiness when you are modeling and acceptance change as you are working because requirements will change over time. You should recognize that incremental change of your system over time enables agility and that you should strive to obtain rapid feedback on your work to ensure that it precisely reflects the needs of your project stakeholders. You should model with a purpose, if you don't know why you are working on something or you don't know what the audience of the model/document actually requires then you shouldn't be working on it. Furthermore, you need numerous models in your intellectual toolkit to be effective. A critical concept is that models are not necessarily documents, a realization that enables you travel light by discarding most of your models once they have fulfilled their purpose. Agile modelers believe that content is more significant than representation, that there are many ways you can model the same concept yet still get it right. To be an effective modeler you need to recognize that open and honest communication is often the best policy to follow to ensure effective teamwork. Finally, a focus on quality work is important because nobody likes to produce sloppy work and that local adaptation of AM to meet the exact needs of your environment is important.

To model in an agile manner you will apply AM's practices as suitable. Fundamental practices comprise creating numerous models in parallel, applying the right artifact(s) for the situation, and iterating to another artifact to continue moving forward at a steady pace. Modeling in small increments, and not attempting to create the magical "all encompassing model" from your ivory tower, is also fundamental to your success as an agile modeler. Because models are only abstract representations of software, abstractions that may not be accurate, you should strive to

Notes

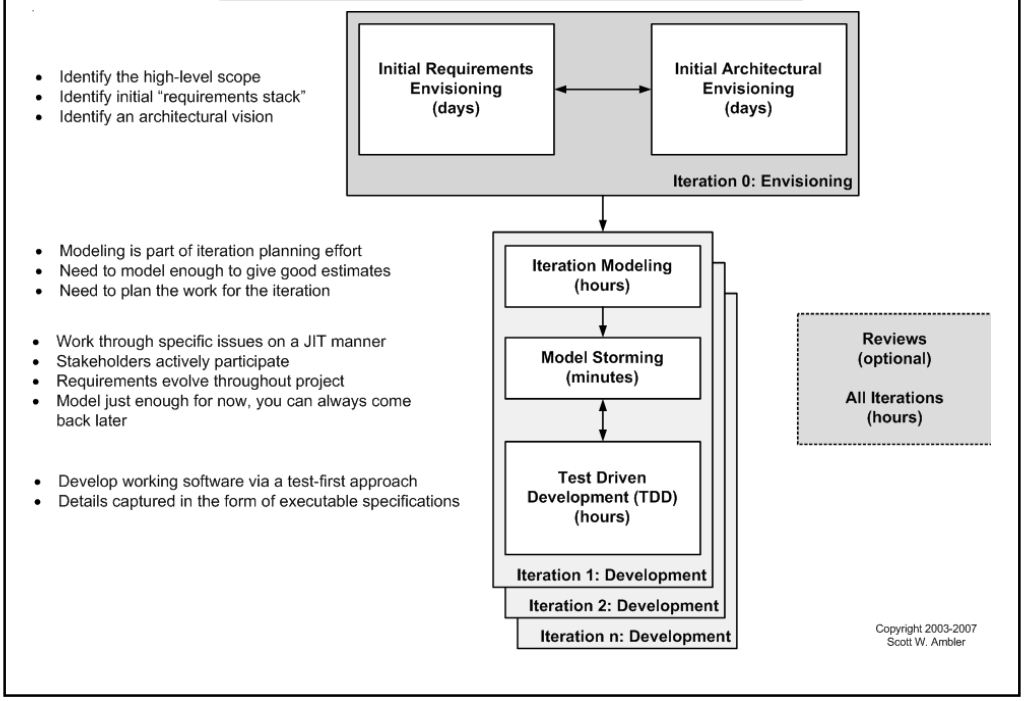
prove it with code to show that your ideas actually work in practice and not just in theory Active stakeholder participation is critical to the success of your modeling efforts because your project stakeholders know what they want and can provide you with the feedback that you require. The principle of assume simplicity is a supported by the practices of creating simple content by focusing only on the aspects that you need to model and not attempting to creating a highly detailed model, depicting models simply via use of simple notations, and using the simplest tools to create your models. You travel light by single sourcing information, discarding temporary models and updating models only when it hurts. Communication is enabled by displaying models publicly, either on a wall or internal web site, through collective ownership of your project artifacts, through applying modeling standards, and by modeling with others. Your development efforts are greatly enhanced when you apply patterns gently. Because you often need to integrate with other systems, including legacy databases as well as web-based services, you will find that you need to formalize contract models with the owners of those systems. Read this article for a better understanding of how AM's practices fit together.

We would argue that AM is an agile approach to modeling, that at its core AM is simply a collection of practices that reflect the principles and values shared by many experienced software developers. With an Agile Model Driven Development (AMDD) (see Figure 5.4) approach you typically do just enough high-level modeling at the beginning of a project to understand the scope and potential architecture of the system, and then during development iterations you do modeling as part of your iteration planning activities and then take a just in time (JIT) model storming approach where you model for several minutes as a precursor to several hours of coding.

Did u know? **What is MDD?**

Model Driven Development (MDD) is an approach to software development where extensive models are created before source code is written.

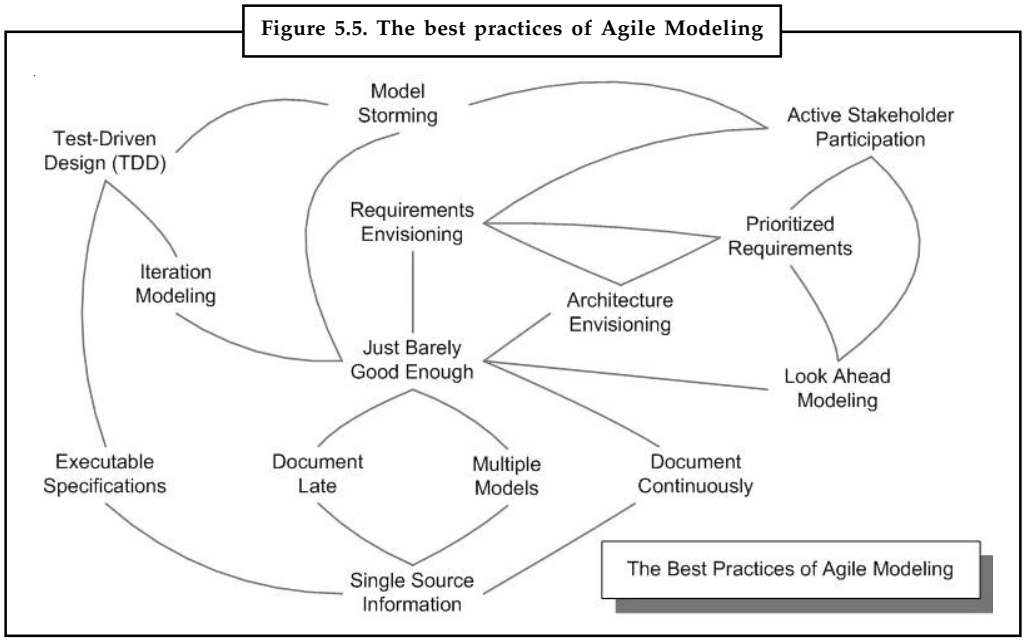
Figure 5.4. Agile Model Driven Development (AMDD)



Notes

Another way to look at Agile Modeling is as a collection of best practices, as you see in Figure 5.5.

Figure 5.5. The best practices of Agile Modeling



Self Assessment

Fill in the blanks:

4. Agile methods break tasks into …….. iterations or parts and do not directly involve long term planning.
5. …… improves a software project in five essential ways; communication, simplicity, feedback, respect, and courage.
6. …….. is scary to traditional organizations because it raises questions about their fundamental world view.
7. …….. is a framework based originally around Rapid Application Development (RAD).
8. DSDM recognizes that projects are limited by time and……, and plans accordingly to meet the business needs.
9. The main criteria for acceptance of …….. in DSDM are on delivering a system that addresses the current business needs.
10. Scrum projects make progress in a series of……., which are time boxed iterations no more than a month long.
11. The …… acts as a liaison between the Product Owner and the team.
12. The …….. methodology is one of the most lightweight, adaptable approaches to software development.
13. Agile Modeling (AM) is a …… methodology for effective modeling and documentation of software-based systems.
14. To model in an agile manner you will apply …… practices as appropriate.

Notes

15. Active stakeholder participation is critical to the success of your modeling efforts because your project …….. know what they want and can provide you with the feedback that you require.

Caselet

Advantage agility!

SAN (Storage Area Network) is more evolved than NAS (Network Attached Storage) says the storage industry. It is more sophisticated, more complex, faster, and more expensive.

The same reason, Network Appliances (NetApp) uses to sell its NAS systems. NAS is simpler, less expensive and equally reliable, the company says, and that makes it "perfect for India," says Anal K. Jain of NetApp.

Network Appliances put all its bets on NAS, until its latest announcement introducing a combined NAS and SAN product. Users can shift from one to the other to access data, a feature that no other vendor in the world offers.

Companies do sell NAS, but the two are separate solutions, not compatible with one another.

NetApp is different not just in that, but also because its solutions are Unix-based. Most traditional storage companies (like EMC) began by offering IBM-compatible disks and memory, and they operated on mainframe-like lines, Jain says.

"There was a high cost of sales, high cost of manufacturing, high margins and the companies were strong on quality and support. Often, support was a good 20 per cent of the cost of the solution itself."

In a break from this, NetApps is lower priced, requires little support and uses Internet Protocol (IP) instead of the more expensive Fibre Channel (FC). Describing it as a "younger, faster moving" company, Jain says that this gives it a huge potential in India, essentially a non-IBM market.

Proprietary systems never took hold in the country and there are practically no mainframes in use. There are a few minis (AS 400s) around, but India hardly has the kind of companies which are described as "enterprise class" that would require mainframes, he believes.

It is partly for this reason that India is one of the three emerging markets identified by NetApps in addition to China and Brazil. "We get out-of-proportion support (from the headquarters in the US)" says Jain.

Despite that, some things still need to be sorted out, such as changing the billing system to take advantage of the duty structure in the country. Today, NetApp ends up costing more in India than its competitors only because of the way it bills. The price advantage available in the rest of the world is not available here.

"There is a perception that NAS is not good for databases, but we have proved them wrong," says Jain. "More than technology rivals, we are fighting perceptions," he says. The other challenge is for the company to build its brand. While it has been named number three in the storage industry by IDC, it is still not well known. The company has to still "start doing things for the Indian market."

Contd...

Notes

It has to grow its headcount from the current nine in the country. It has three partners, including Wipro and Apara. NetApp has a "few dozen customers" in India, including GE, ICICI, Cisco, Kotak, and Synopsis.

5.3 Summary

- To become Agile, most organizations will need to change their perspective. Most of our assumptions about business, about technology and organizations are at least 50 years old. They have outlived their time.
- As a result, we are preaching, teaching, and practicing policies that are increasingly at odds with reality and therefore counterproductive.
- Computer science is a young science.
- That training dictated how we approached software development for an entire generation.
- But now after decades of building software to be expensive, unwanted, and unreliable we have come to realize software is different.
- Extreme Programming is successful because it stresses customer satisfaction. Instead of delivering everything you could possibly want on some date far in the future this process delivers the software you need as you need it.
- DSDM was developed in the United Kingdom in the 1990s by a consortium of vendors and experts in the field of Information System (IS) development, the DSDM Consortium, combining their best-practice experiences.
- The scrum team is self-organizing in that there is no overall team leader who decides which person will do which task or how a problem will be solved.
- AM is based on a collection of principles, such as the importance of assuming simplicity when you are modeling and embracing change as you are working because requirements will change over time.

5.4 Keywords

AMDD: Model Driven Development

AM: Agile Modeling

DSDM: Dynamic Systems Development Method

FDD: Feature-Driven Development

IS: Information System

JIT: Just in time

5.5 Review Questions

1. Agility means being responsive or flexible within a defined context. Explain.
2. Is "Agile process model" refers to a software development approach based on iterative development? Why or why not?

Notes

3. "Most models promote a business approach that maps software development with customer needs and company goals." Discuss.
4. Extreme Programming is successful because it stresses customer satisfaction. Do you agree? Give reasons and examples to support your answer.
5. In complex environments, planning is a paradox. Justify this statement with examples.
6. Substantiate why managing in a complex environment is scary as hell — it is also a blast?
7. Collaborative activities build products. Discuss this statement with diagram.
8. Explain the concept of RAD? Why RAD is an anathema to many IT organizations. Give reasons.
9. In the Scrum methodology why the team is responsible for completing work? Give reasons.
10. FDD is a model-driven, short-iteration process. Do you think so? Why?

Answers: Self Assessment

1.	initiative	2.	business
3.	conformance	4.	smaller
5.	Extreme Programming	6.	RAD
7.	Dynamic Systems Development Method (DSDM)		
8.	Resources	9.	Deliverable
10.	Sprints	11.	ScrumMaster
12.	Crystal	13.	practice-based
14.	AM's	15.	stakeholders

5.6 Further Readings

Online links

http://www.agilemodeling.com/essays/amdd.htm

http://searchsoftwarequality.techtarget.com/definition/Scrum

Notes

Unit 6: Requirement Engineering

CONTENTS

Objectives

Introduction

6.1 Requirement Engineering

6.2 A Brigade of Design & Construction

6.3 Requirements Engineering Tasks

6.3.1 Requirements Reviews/Inspections

6.3.2 Elicitation

6.3.3 Requirement Negotiation

6.3.4 Requirement Specification

6.3.5 Requirements Validation

6.3.6 Requirements Management

6.4 Summary

6.5 Keywords

6.6 Review Questions

6.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize a brigade to design and construction
- Describe requirement engineering task
- Explain requirement management

Introduction

Requirements specify the "what" of a system, not the "how". Requirements engineering provides the appropriate mechanism to understand what the customer desires, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system. Thus, requirements engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behavior and its associated constraints.

6.1 Requirement Engineering

The requirements engineering includes the following activities:

- Identification and documentation of customer and user's needs.

Notes

- Creation of a document describing the external behavior and associated constraints that will satisfy those needs.
- Analysis and validation of the requirements document to ensure consistency, completeness and feasibility.
- Evolution of needs.

The primary output of requirements engineering is requirements specification. If it describes both hardware and software it is a system requirement and if it describes only software requirement it is a software requirements specification. In both these cases, the system is treated as a black box.

Requirements Specification

Requirements specification can be done in the form of a document, graphical model, prototype or a combination of all these. System specification can be elicited using a standard template or a flexible approach depending upon the system being developed.

System specification is produced as a result of system and requirements engineering. It contains details about the hardware, software, database etc. It describes both the function and constraints required for developing a system. It also tells about the input and output information from the system.

Requirements Validation

The work product produced as a result of requirements engineering is checked for quality during the validation step. Requirements validation ensures that all the requirements have been stated correctly, inconsistencies and errors have been detected and corrected and that the work product conforms to the standard for process, project and product.

Although the requirements validation can be done in any way that leads to discovery of errors, the requirements can be examined against a checklist. Some of the checklist questions can be:

- Are the requirements clear or they can be misinterpreted?
- Is the source of the requirements identified?
- Has the final statement of requirement been verified against the source?
- Is the requirement quantitatively bounded?
- Is the requirement testable?
- Does the requirement violate domain constraints?

These questions and their likes ensure that validation team does everything possible for a thorough review.

Requirements Management

Requirements may change throughout the life of a computer based system. Requirements management includes the activities that help a project team to identify, control and track requirements and changes to these requirements at any time during the project.

Requirements management begins with identification. Once the requirements have been identified traceability tables are created. This table relates each requirement to one or more aspects of system or its environment. They may be at times used as a requirements database in order to understand how a change in one requirement will affect other aspects of the system being built.

Notes

Did u know? **What is the exact meaning of software requirement engineering?**

Software requirements engineering is the process of determining what is to be produced in a software system. In developing a complex software system, the requirements engineering process has the widely recognized goal of determining the needs for, and the intended external behavior, of a system design.

6.2 A Brigade of Design & Construction

The final goal of any engineering activity is to create some kind of documentation. When ea design effort is complete, the design documentation is given to the manufacturing team. This is a different set of people with a different set of skills from those of the design team. If the design documents truly represent a complete design, the manufacturing team can proceed to build the product. In fact, they can proceed to build much of the product without further assistance from the designers. After reviewing the software development life cycle today, it appears that the only software documentation that actually seems to satisfy the criteria of an engineering design are the source code listings.

Software runs on computers. It is a sequence of ones and zeros. Software is not a program listing. A program listing is a document that represents a software design. Compilers, linkers, and interpreters actually build (manufacture) software. Software is very cheap to build. You just press a button. Software is expensive to design because it is complicated and all phases of the development cycle are part of the design process. Programming is a design activity; a good software design process recognizes this and does not hesitate to code when coding makes sense. Coding actually makes sense more often than believed. Often the process of rendering the design in code will reveal oversights and the need for more design effort. The earlier this happens, the better. Testing and debugging are design activities – they are the equivalents of design validation and refinement in other engineering disciplines. They can not be short changed. Formal validation methods are not of much use because it is cheaper to build software and test it than to prove it.

Think about it. When you are programming, you are doing detailed design. The manufacturing team for software is your compiler or interpreter. The source is the only complete specification of what the software will do.

Whenever objects are created for people to use, design is pervasive. Design may be done methodically or offhandedly, consciously or accidentally. But when people create software – or any other product – decisions are made and objects are constructed that carry with them an intention of what those objects will do and how they will be perceived and used.

The education of computer professionals has often concentrated on the understanding of computational mechanisms, and on engineering methods that are intended to ensure that the mechanisms behave as the programmer intends. The focus is on the objects being designed: the hardware and software. The primary concern is to implement a specified functionality efficiently. When software engineers or programmers say that a piece of software works, they typically mean that it is robust, is reliable, and meets its functional specification. These concerns are indeed important. Any designer who ignores them does so at the risk of disaster.

But this inward-looking perspective, with its focus on function and construction, is one-sided. To design software that really works, we need to move from a constructor's-eye view to a designer's-eye view, taking the system, the users, and the context all together as a starting point.

Notes

Notes Good design produces an object that works for people in a context of values and needs, to produce quality results and a satisfying experience.

Self Assessment

Fill in the blanks:

1. The primary output of requirements engineering is requirements……...
2. …….. specification can be elicited using a standard template or a flexible approach depending upon the system being developed.
3. The work product produced as a result of requirements engineering is checked for quality during the ……. step.
4. Requirements management begins with…….
5. Software runs on computers. It is a sequence of……..
6. Requirements engineering is an important aspect of any……., and is a general term used to encompass all the activities related to requirements.

6.3 Requirements Engineering Tasks

The processes used for RE vary widely depending on the application domain, the people involved and the organization developing the requirements.

However, there are a number of generic activities common to most processes:

- Inception
- Requirements elicitation
- Negotiation
- Requirements specification
- Requirements validation
- Requirement Management

6.3.1 Requirements Reviews/Inspections

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews. (Stakeholders)
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review Check-list

- ***Verifiability:*** Is the requirement realistically testable?
- ***Comprehensibility:*** Is the requirement properly understood?
- ***Traceability:*** Is the origin of the requirement clearly stated?

Notes

- *Adaptability:* Can the requirement be changed with minimum impact on other requirements? (Especially when change is anticipated!)

6.3.2 Elicitation

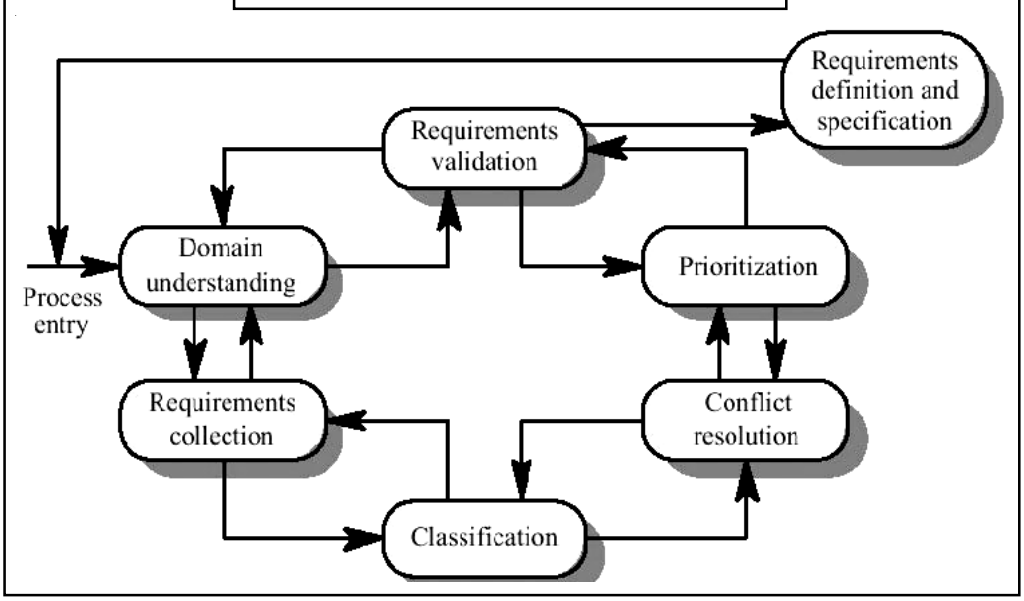
- Involves working with customers to learn about the application domain, the services needed and the system's operational constraints.
- May also involve end-users, managers, maintenance personnel, domain experts, trade unions, etc. (That is, any stakeholders.)

Problems of Elicitation and Analysis

- Getting all, and only, the right people involved.
- Stakeholders often don't know what they really want ("I'll know when I see it").
- Stakeholders express requirements in their own terms.
- Stakeholders may have conflicting requirements.
- Requirements change during the analysis process.
- Organizational and political factors may influence the system requirements.

Task Requirements change during the analysis process. In a group of four analyze the various ways to overcome this problem.

Figure 6.1: The Elicitation and Analysis Process



Viewpoint-oriented Elicitation

- Stakeholders represent different ways of looking at a problem (viewpoints)

Notes

- A multi-perspective analysis is important, as there is no single correct way to analyze system requirements.
- Provides a natural way to structure the elicitation process.

Types of Viewpoints

- ***Data sources or sinks:*** Viewpoints are responsible for producing or consuming data. Analysis involves checking that assumptions about sources and sinks are valid.
- ***Representation frameworks:*** Viewpoints represented by different system models (i.e., dataflow, ER, finite state machine, etc.). Each model yields different insights into the system.
- ***Receivers of services:*** Viewpoints are external to the system and receive services from it. Natural to think of end-users as external service receivers.

Method-based RE

"Structured methods" to elicit, analyse, and document requirements.

Examples include:

- Ross' Structured Analysis (SA),
- Volere Requirements Process (www.volere.co.uk).
- Knowledge Acquisition and Sharing for Requirement Engineering (KARE) (www.kare.org),
- Somerville's Viewpoint-Oriented Requirements Definition (VORD), and
- The baut's Scenario-Based Requirements Engineering (SBRE)˜

Scenarios

- Depict examples or scripts of possible system behavior.
- People often relate to these more readily than to abstract statements of requirements.
- Particularly useful in dealing with fragmentary, incomplete, or conflicting requirements.

Scenario Descriptions

- System state at the beginning of the scenario.
- Sequence of events for a specific case of some generic task the system is required to accomplish.
- Any relevant concurrent activities.
- System state at the completion of the scenario.

Scenario-Based Requirements Engineering (SBRE)

- Marcel support environment allows rapid construction of an operational specification of the desired system and its environment.
- Based on a forward chaining rule-based language.

Notes

- An interpreter executes the specification to produce natural language based scenarios of system behavior.

UML Use-cases and Sequence Diagrams

- Use-cases are a graphical notation for representing abstract scenarios in the UML.
- They identify the actors in an interaction and describe the interaction itself.
- A set of use-cases should describe all types of interactions with the system.
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing.

Figure 6.2: Library Use-cases

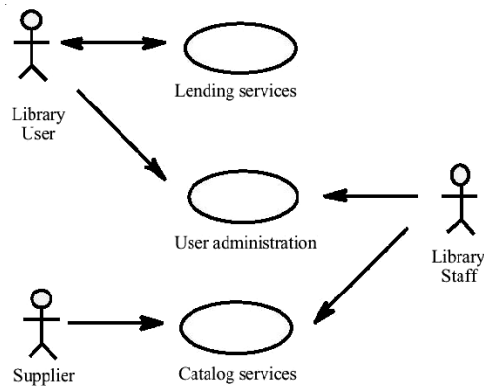
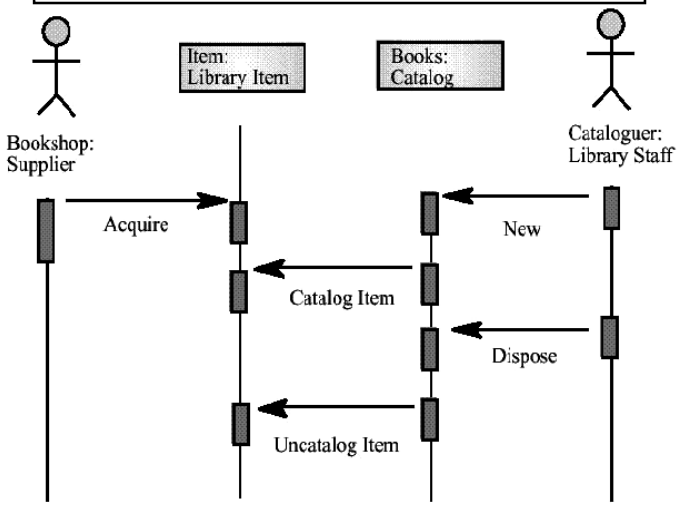


Figure 6.3: Catalogue Management Sequence Diagram



Social and Organizational Factors

- All software systems are used in a social and organizational context. This can influence or even dominate the system requirements.

Notes

- Good analysts must be sensitive to these factors, but there is currently no systematic way to tackle their analysis.

Example:

1. Consider a system, which allows senior management to access information without going through middle managers.
2. Managerial status. Senior managers may feel that they are too important to use a keyboard. This may limit the type of system interface used.
3. Managerial responsibilities. Managers may have no uninterrupted time when they can learn to use the system
4. Organizational resistance. Middle managers who will be made redundant may deliberately provide misleading or incomplete information so that the system will fail.

Ethnography

- Social scientists spends considerable time observing and analyzing how people actually work.
- People do not have to explain or articulate what they do.
- Social and organizational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Focused Ethnography

- Developed during a project studying the air traffic control process.
- Combines ethnography with prototyping.
- Prototype development raises issues, which focus the ethnographic analysis.
- Problem with ethnography alone: it studies existing practices, which may not be relevant when a new system is put into place.

6.3.3 Requirement Negotiation

After they have been collected, requirements must be analyzed to obtain a satisfactory understanding of the customer's need and negotiated to establish an agreed set of consistent (unambiguous, correct, complete, etc.) requirements.

The requirements negotiation process is expensive and time-consuming, requiring very experienced personnel exercising considerable judgment. This is exacerbated by the fact that the requirements negotiation process is not sufficiently structured that an automated requirements negotiation methodology would be appropriate.

6.3.4 Requirement Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. Typical physical systems have a relatively small

Notes

number of such values. Typical software has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements.

Caution So, in software engineering jargon, "software requirements specification" typically refers to the production of a document, or its electronic equivalent, which can be systematically reviewed, evaluated, and approved.

For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required.

There are some approaches to requirements specification:

- Natural language
- Structured natural language
- Design description language
- Requirements specification language
- Graphical notation
- Formal specification

6.3.5 Requirements Validation

- Concerned with whether or not the requirements define a system that the customer really wants.
- Requirements error costs are high, so validation is very important. (Fixing a requirements error after delivery may cost up to 100 times that of fixing an error during implementation.)

Requirements Checking

- ***Validity:*** Does the system provide the functions which best support the customer's needs?
- ***Consistency:*** Are there any requirements conflicts?
- ***Completeness:*** Are all functions required by the customer included?
- ***Realism:*** Can the requirements be implemented given available budget and technology?
- ***Verifiability:*** Can the requirements be tested?

Requirements Validation Techniques

- ***Requirements reviews/inspections:*** systematic manual analysis of the requirements.
- ***Prototyping:*** using an executable model of the system to check requirements.
- ***Test-case generation:*** developing tests for requirements to check testability.
- ***Automated consistency analysis:*** checking the consistency of a structured requirements description.

Notes

6.3.6 Requirements Management

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- New requirements emerge during the process as business needs change and a better understanding of the system is developed.
- The priority of requirements from different viewpoints changes during the development process.
- The business and technical environment of the system changes during its development.

Enduring and Volatile Requirements

- ***Enduring requirements:*** Stable requirements derived from the core activity of the customer organization. For example, a hospital will always have doctors, nurses, etc. May be derived from domain models.
- ***Volatile requirements:*** Requirements which change during development or when the system is in use. E.g., requirements derived from the latest health-care policy.

Classification of Requirements

- ***Mutable requirements:*** those that change due to changes in the system's environment.
- ***Emergent requirements:*** those that emerge as understanding of the system develops.
- ***Consequential requirements:*** those that result from the introduction of the system.
- ***Compatibility requirements:*** those that depend on other systems or organizational processes.

Requirements Management Planning

- During requirements management planning, you must decide on:
  - *Requirements identification*: How requirements will be individually identified.
  - *A change management process*: A process to be followed when analyzing the impact and costs of a requirements change.
  - *Traceability policies*: The amount of information about requirements relationships that is maintained.
  - *CASE tool support*: The tool support required to help manage requirements change.

Traceability

- Traceability is concerned with the relationships between requirements, their sources, and the system design.
- Source traceability – links from requirements to stakeholders who proposed these requirements.
- Requirements traceability – links between dependent requirements.
- Design traceability – links from the requirements to the design.

Notes

CASE Tool Support

- Requirements storage – requirements should be managed in a secure, managed data store.
- Change management – the process of change management is a workflow process whose stages can be defined and information flow between the stages partially automated.
- Traceability management – automated discovery and documentation of relationships between requirements

Requirements change management should apply to all proposed changes to the requirements.

Principal Stages

- Problem analysis – discuss identified requirements problem and propose specific changes.
- Change analysis and costing – assess effects of change on other requirements.
- Change implementation – modify requirements document and others to reflect change.

Task Discuss an example of a type of system where social and political factors might strongly influence the system requirements. Explain why these factors are important in your example.

Self Assessment

Fill in the blanks:

7. Analysis involves checking that assumptions about sources and sinks are …….
8. A set of …… should describe all types of interactions with the system.
9. ……. diagrams may be used to add detail to use-cases by showing the sequence of event processing.
10. The requirements ……. process is expensive and time-consuming, requiring very experienced personnel exercising considerable judgment.
11. Requirements error costs are high, so …….. is very important.
12. New …….. emerge during the process as business needs change and a better understanding of the system is developed.
13. The process of change management is a …….. process whose stages can be defined and information flow between the stages partially automated.
14. The ……. support required to help manage requirements change.
15. ……. means using an executable model of the system to check requirements.

Notes

Caselet

Securing India's Science Future

In 2004, while reviewing the science and technology policy of the Government of India, A.P.J. Abdul Kalam said: "In a world where the powers are determined by the share of the world's knowledge, reflected by patents, papers and so on…it is important for India to put all her acts together to become a continuous innovator and creator of science and technology intensive products." The importance of scientific and technological advancement in today's highly globalised environment cannot be overstated.

If we are to go by an observation in a report by India's Department of Foreign Affairs and Trade that "the health of a nation depends, among other factors, on the health of the state of its science and technology," we have cause to be concerned about the health of our nation. In an increasingly competitive global economy, knowledge-driven growth powered by innovation is a critical imperative. While India is uniquely positioned to use technology for progress, it has in the recent past lagged behind considerably in the quality and spread of science research. This is a critical lacuna that could well determine the fate not just of our scientific and developmental future but, more importantly, of our progress as a nation.

Status and Challenges

A recent study by Thomson Reuters titled Global Research Report: India concluded that, given ideal conditions, India's research productivity would be on a par with that of most G8 nations within seven to eight years and that it could probably overtake them in 2015-2020. In the last decade, India has seen its annual output of scientific publications grow from roughly 16,500 in 1998 to nearly 30,000 in 2007. Before we pat ourselves on the back, it would be good to consider things in perspective. Although India produces about 400,000 engineering graduates and about 300,000 computer science graduates every year, just about 20,000 master's degree holders and fewer than 1,000 PhDs in engineering graduate each year.

In 2007-08, India had about 156 researchers per million in the population, compared with 4,700 per million in the United States. In terms of sheer numbers, in 2007 China had 1,423,000 researchers, second internationally to the United States, which had almost 1,571,000. India by comparison had 154,800. India's spend on R&D in 2007-08 was about US$ 24 billion compared with China's investment of about US$ 104 billion back in 2006 and the United States' US$ 368 billion. These comparative allocations, which have not changed much since then, reveal the gross inadequacy in India's commitment to research, considering our scientists' potential and our aspirations as a nation.

A survey of 47 universities conducted by the University Grants Commission in 2007-08 revealed vacancy levels as high as 51 per cent. It is evident that the majority of India's graduating engineers, particularly the cream, are going directly into the job market – affecting the number and quality of those available for research. This trend is partly because of the widespread notion that remuneration in a research career is below par and partly because of the lack of adequate encouragement and direction for young potential researchers.

Not enough PhDs graduate in India — be it in number or excellence — to meet the growing staff requirements of its universities. As a result, even the quality of faculty has shown a declining trend and this is bound to have serious repercussions on the country's intellectual edge. Add to this the issues of politicisation of the Indian scientific establishment,

Contd...

Notes

particularly in according due recognition, the lack of adequate funding and infrastructure, and the disparity in research funds and facilities available to universities. Further, the long-time policy of target-oriented research in selected thrust areas, as against open-ended research, has often come at the cost of the basic sciences. It is common knowledge that research in basic sciences is a critical prerequisite for the success of applied sciences and the bedrock of all technological advancement.

Way Forward

The key to continued success for India in a globalised knowledge-driven economy is building a higher education system that is superior in quality and committed encouragement of relevant research in science and technology. What is needed is an environment where the government, universities, companies, venture capitalists, and other stakeholders come together for the enablement of the entire science ecosystem with an eye on future sustainability.

A manifold increase in the country's investment in scientific research is only the beginning. The government must play a key role by enhancing the number, quality, and management of science schools focussed on science research. Given the present government's direction, this is something that could come to pass over the next few years. The IIT model of success needs to be replicated on a far larger scale. Providing the requisite autonomy to research institutions is an important necessity. Professors, scientists, and institution heads are often the people best informed on the necessary conditions required for the advancement of research goals. They must be enabled with the autonomy to create those conditions.

With industry often being the downstream beneficiary of several research efforts, increased interaction between industry and research establishments is important. There needs to be a sound incentive system for the corporate sector involved in scientific R&D as well, with infrastructure and financial benefits, as is the case with the IT industry. This includes viable incentives such as tax breaks for corporate R&D efforts and special economic zones and technology parks for R&D establishments.

In an age where issues of research interest are often global in nature, we must encourage active interaction and exchange with international research institutions. Cross-continental research cooperation and knowledge sharing was at the base of the story of the three winners of the Chemistry Nobel for 2009, among who was Dr Venkataraman Ramakrishnan, of Indian origin. Creating partnerships with relevant peer institutions in India and abroad, hosting events and conferences and getting eminent researchers and scientists to shed light on progress in key research areas and supporting related publications — these measures will go a long way not only in enriching India's research eco-system but also in enticing potential young researchers to the cause.

For instance, it should be a practice to invite Nobel Prize winners or similar eminent international scientists for seminars with selected young research minds and students in India at least annually. This could prove a valuable source of insight and inspiration to young potentials. In this regard, I would suggest the creation of an institution like the National Science Foundation, endowed with a suitable corpus from key public and private stakeholders, conferred with the responsibility of regularly undertaking such initiatives.

The importance of rewards and recognition for scientific research cannot be understated as a measure to encourage talented youngsters to consider careers in research. There is a need to recognise and applaud the accomplishments of our researchers and scientists, just as we applaud the achievements of our sportspersons. This must include measures such as financial support to encourage students to adopt research careers, suitable incentives, and

Contd...

Notes

awards for scientific achievement. For example, the best and brightest top 1000 students across India with the potential and inclination for research could be provided guaranteed funding for their further education and early careers in research. And the private sector could play the role of a patron here.

The government of Prime Minister Manmohan Singh has avowed goals to reduce poverty and stimulate development. One among the many facilitators for this is a focussed investment in science and technology, which the Prime Minister has acknowledged by announcing a doubling of related spend in terms of percentage of GDP over the next couple of years. Parliament's approval for the creation of a National Science and Engineering Research Board, responsible for funding and furthering scientific research, is laudable and a significant step in the right direction. The Human Resource Development Ministry's efforts to improve the higher education system and the establishment of five new Indian Institutes of Science Education and Research in the past three years will provide a vital boost to the cause of scientific research in India.

The need for a strong science ecosystem based on a sound research foundation has an integral connect with India's development as a world power. India needs the best intellect available for government, business, military, or any aspect of society to strive for global excellence. Their accomplishments need to be lauded and brought to the forefront. Globally, there are several prestigious awards that ensure due recognition, visibility, and reward for outstanding achievements in research. There is a need to emulate this in India to encourage higher levels of research work with an impact on India's development. One such initiative is the Infosys Prize, which has been instituted to honour outstanding researchers who make a difference to India's scientific future and to motivate youngsters to consider careers in research. Overall, there is a need for many more integrated, multi-pronged, and multi-institutional interventions to encourage greater participation and strengthening of scientific research in India. Our success in ensuring this over the next few years will determine how best we will be able to secure India's scientific and developmental future.

6.4 Summary

- The requirements engineering process includes a feasibility study, elicitation and analysis, specification, and validation.
- Requirements analysis is an iterative process involving domain understanding, requirements collection, classification, structuring, prioritization and validation.
- Systems have multiple stakeholders with different viewpoints and requirements. Social and organization factors influence system requirements.
- Requirements validation is concerned with checks for validity, consistency, completeness, realism, and verifiability.
- Business, organizational, and technical changes inevitably lead to changing requirements. Requirements management involves careful planning and a change management process.

6.5 Keywords

KARE: Knowledge Acquisition and Sharing for Requirement Engineering

SA: Structured Analysis

SBRE: Scenario-Based Requirements Engineering

VORD: Viewpoint-Oriented Requirements Definition

Notes

6.6 Review Questions

1. What are the feasibility study issues?
2. Suggest who might be stakeholders in a university student records system. Explain why it is almost inevitable that the requirements of different stakeholders will conflict in some ways.
3. For three of the viewpoints identified in the library cataloguing system, suggest services which might be provided to that viewpoint, data which the viewpoint might provide and events which control the delivery of these services.
4. Using your own knowledge of how an ATM is used, develop a set of use cases that could be used to derive the requirements for an ATM system.
5. How would the organization cope if the system weren't implemented?
6. What are the current process problems and how would the system help with these?
7. Is new technology needed New skills?
8. What must be supported by the system and what had not be supported?
9. Requirements may change throughout the life of a computer based system. Explain.
10. Requirements management is the process of managing changing requirements during the requirements engineering process and system development. Discuss.

Answers: Self Assessment

1.	specification	2.	System
3.	validation	4.	identification
5.	ones and zeros	6.	software project
7.	valid	8.	use-cases
9.	Sequence	10.	negotiation
11.	validation	12.	requirements
13.	workflow	14.	CASE tool
15.	Prototyping		

6.7 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering*, Second Edition, PHI.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering a Practioner's Approach*, 5th edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering*, Sixth edition, Pearson Education.

Online links

http://ksi.cpsc.ucalgary.ca/KAW/KAW96/herlea/FINAL.html

http://en.wikipedia.org/wiki/Requirements_analysis

Notes

Unit 7: Software Engineering Practice

CONTENTS

Objectives

Introduction

7.1 The Essence of Practice

7.2 Core Principles

7.3 Planning Practices

7.4 Modeling Practices

7.4.1 Analysis Modeling Principles

7.4.2 Design Modeling Principles

7.5 Construction Practice

7.5.1 Coding Principle

7.5.2 Coding Concepts

7.5.3 Testing Principles

7.6 Summary

7.7 Keywords

7.8 Review Questions

7.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the essence of software practices
- Describe software engineering core principles
- Examine planning principles
- Recognize modeling and construction practices

Introduction

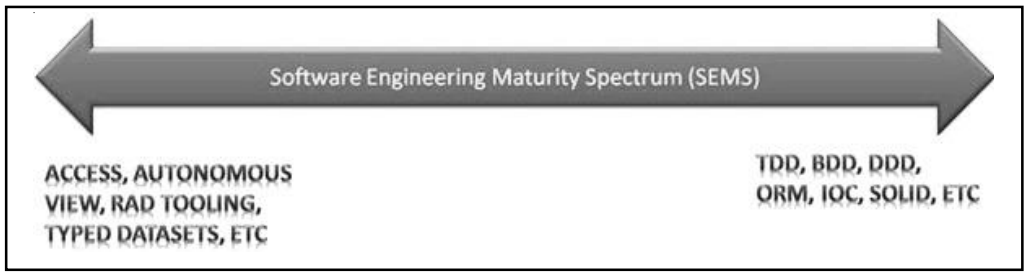
This unit covers the various software engineering practices and principles. In this unit, we will read the core principles of software engineering, the planning practices, modeling practices, analysis and design modeling practices and the software construction practices

7.1 The Essence of Practice

If you compare any two software projects (specifically comparing their codebases) you'll often see very different levels of maturity in the software engineering practices employed. By software engineering practices, we specifically refer to the quality of the code and the amount of technical debt present in the project.

Notes

Things such as Test Driven Development, Domain Driven Design, Behavior Driven Development, proper adherence to the SOLID principles, etc. are all practices that you would expect at the mature end of the spectrum. At the other end of the spectrum would be the quick-and-dirty solutions that are done using something like an Access Database, Excel Spreadsheet, or maybe some quick "drag-and-drop coding".



We believe there is a time and a place for projects at every part of that SEMS. The risks and costs associated with under-engineering solutions have been written about a million times over so we won't bother going into them again here, but there are also (unnecessary) costs with over-engineering a solution. Sometimes putting multiple layers, and IoC containers, and abstracting out the persistence, etc. is complete overkill if a one-time use Access database could solve the problem perfectly well.

A lot of software developers we talk to seem to automatically jump to the very right-hand side of this SEMS in everything they do. A common rationalization we hear is that it may seem like a small trivial application today, but these things always grow and stick around for many years, then you're stuck maintaining a big ball of mud. We think this is a cop-out. Sure you can't always anticipate how an application will be used or grow over its lifetime (can you ever??), but that doesn't mean you can't manage it and evolve the underlying software architecture as necessary (even if that means having to toss the code out and rewrite it at some point…maybe even multiple times).

Our thoughts are that we should be making a conscious decision around the start of each project approximately where on the SEMS we want the project to exist. We believe this decision should be based on 3 factors:

1. ***Importance:*** How important to the business is this application? What is the impact if the application were to suddenly stop working?
2. ***Complexity:*** How complex is the application functionality?
3. ***Life-Expectancy:*** How long is this application expected to be in use? Is this a one-time use application, does it fill a short-term need, or is it more strategic and is expected to be in-use for many years to come?

Of course this isn't an exact science. You can't say that Project X should be at the 73% mark on the SEMS and expect that to be helpful. Our point is not that you need to precisely figure out what point on the SEMS the project should be at then translate that into some prescriptive set of practices and techniques you should be using. Rather our point is that we need to be aware that there is a spectrum, and that not everything is going to be (or should be) at the edges of that spectrum, indeed a large number of projects should probably fall somewhere within the middle; and different projects should adopt a different level of software engineering practices and maturity levels based on the needs of that project.

Notes

To give an example of this way of thinking from our day job:

Every couple of years my company plans and hosts a large event where ~400 of our customers all fly in to one location for a multi-day event with various activities. We have some staff whose job it is to organize the logistics of this event, which includes tracking which flights everybody is booked on, arranging for transportation to/from airports, arranging for hotel rooms, name tags, etc. The last time we arranged this event all these various pieces of data were tracked in separate spreadsheets and reconciliation and cross-referencing of all the data was literally done by hand using printed copies of the spreadsheets and several people sitting around a table going down each list row by row.

Notes Obviously there is some room for improvement in how we are using software to manage the event's logistics.

The next time this event occurs we plan to provide the event planning staff with a more intelligent tool (either an Excel spreadsheet or probably an Access database) that can track all the information in one location and make sure that the various pieces of data are properly linked together (so for example if a person cancels you only need to delete them from one place, and not a dozen separate lists). This solution would fall at or near the very left end of the SEMS meaning that we will just quickly create something with very little attention paid to using mature software engineering practices. If we examine this project against the 3 criteria we have listed above for determining it's place within the SEMS we can see why:

- ***Importance:*** If this application were to stop working the business doesn't grind to a halt, revenue doesn't stop, and in fact our customers wouldn't even notice since it isn't a customer facing application. The impact would simply be more work for our event planning staff as they revert back to the previous way of doing things (assuming we don't have any data loss).
- ***Complexity:*** The use cases for this project are pretty straightforward. It simply needs to manage several lists of data, and link them together appropriately. Precisely the task that access (and/or Excel) can do with minimal custom development required.
- ***Life-Expectancy:*** For this specific project we're only planning to create something to be used for the one event (we only hold these events every 2 years).

Let's assume we hack something out quickly and it works great when we plan the next event. We may decide that we want to make some tweaks to the tool and adopt it for planning all future events of this nature. In that case we should examine where the current application is on the SEMS, and make a conscious decision whether something needs to be done to move it further to the right based on the new objectives and goals for this application. This may mean scrapping the access database and re-writing it as an actual web or windows application. In this case, the life-expectancy changed, but let's assume the importance and complexity didn't change all that much. We can still probably get away with not adopting a lot of the so-called "best practices".

Example: We can probably still use some of the RAD tooling available and might have an Autonomous style design that connects directly to the database and binds to typed datasets (we might even choose to simply leave it as an access database and continue using it; this is a decision that needs to be made on a case-by-case basis).

At Anvil Digital we have aspirations to become a primarily product-based company. So let's say we use this tool to plan a handful of events internally, and everybody loves it. Maybe a couple years down the road we decide we want to package the tool up and sell it as a product to

Notes

some of our customers. In this case the project objectives/goals change quite drastically. Now the tool becomes a source of revenue, and the impact of it suddenly stopping working is significantly less acceptable. Also as we hold focus groups, and gather feedback from customers and potential customers there's a pretty good chance the feature-set and complexity will have to grow considerably from when we were using it only internally for planning a small handful of events for one company.

In this fictional scenario we would expect the target on the SEMS to jump to the far right. Depending on how we implemented the previous release we may be able to refactor and evolve the existing codebase to introduce a more layered architecture, a robust set of automated tests, introduce a proper ORM and IoC container, etc. More likely in this example the jump along the SEMS would be so large we'd probably end up scrapping the current code and re-writing. Although, if it was a slow phased roll-out to only a handful of customers, where we collected feedback, made some tweaks, and then rolled out to a couple more customers, we may be able to slowly refactor and evolve the code over time rather than tossing it out and starting from scratch.

The key point we are trying to get across is not that you should be throwing out your code and starting from scratch all the time. But rather that you should be aware of when and how the context and objectives around a project changes and periodically reassess where the project currently falls on the SEMS and whether that needs to be adjusted based on changing needs.

Notes There is also the idea of "spectrum decay". Since our industry is rapidly evolving, what we currently accept as mature software engineering practices (the right end of the SEMS) probably won't be the same 3 years from now. If you have a project that you were to assess at somewhere around the 80% mark on the SEMS today, but don't touch the code for 3 years and come back and reassess its position, it will almost certainly have changed since the right end of the SEMS will have moved farther out (maybe the project is now only around 60% due to decay).

7.2 Core Principles

Separation of Concerns

Separation of concerns is recognition of the need for human beings to work within a limited context. The human mind is limited to dealing with approximately seven units of data at a time. A unit is something that a person has learned to deal with as a whole – a single abstraction or concept. Although human capacity for forming abstractions appears to be unlimited, it takes time and repetitive use for an abstraction to become a useful tool; that is, to serve as a unit.

When specifying the behavior of a data structure component, there are often two concerns that need to be dealt with: Basic functionality and support for data integrity. A data structure component is often easier to use if these two concerns are divided as much as possible into separate sets of client functions. It is certainly helpful to clients if the client documentation treats the two concerns separately. Further, implementation documentation and algorithm descriptions can profit from separate treatment of basic algorithms and modifications for data integrity and exception handling.

There is another reason for the importance of separation of concerns. Software engineers must deal with complex values in attempting to optimize the quality of a product. From the study of algorithmic complexity, we can learn an important lesson. There are often efficient algorithms

Notes

for optimizing a single measurable quantity, but problems requiring optimization of a combination of quantities are almost always NP-complete. Although it is not a proven fact, most experts in complexity theory believe that NP-complete problems cannot be solved by algorithms that run in polynomial time.

In view of this, it makes sense to separate handling of different values. This can be done either by dealing with different values at different times in the software development process, or by structuring the design so that responsibility for achieving different values is assigned to different components.

As an example of this, run-time efficiency is one value that frequently conflicts with other software values. For this reason, most software engineers recommend dealing with efficiency as a separate concern. After the software is design to meet other criteria, it's run time can be checked and analysed to see where the time is being spent.

Caution If necessary, the portions of code that are using the greatest part of the runtime can be modified to improve the runtime.

Modularity

The principle of modularity is a specialization of the principle of separation of concerns. Following the principle of modularity implies separating software into components according to functionality and responsibility.

Abstraction

The principle of abstraction is another specialization of the principle of separation of concerns. Following the principle of abstraction implies separating the behavior of software components from their implementation. It requires learning to look at software and software components from two points of view: what it does, and how it does it.

Failure to separate behavior from implementation is a common cause of unnecessary coupling. For example, it is common in recursive algorithms to introduce extra parameters to make the recursion work. When this is done, the recursion should be called through a non-recursive shell that provides the proper initial values for the extra parameters. Otherwise, the caller must deal with a more complex behavior that requires specifying the extra parameters. If the implementation is later converted to a non-recursive algorithm then the client code will also need to be changed.

Task In a group of four explain that what you have understood by the statement. "Design by contract is an important methodology for dealing with abstraction."

Anticipation of Change

Computer software is an automated solution to a problem. The problem arises in some context, or domain that is familiar to the users of the software. The domain defines the types of data that the users need to work with and relationships between the types of data.

Software developers, on the other hand, are familiar with a technology that deals with data in an abstract way. They deal with structures and algorithms without regard for the meaning or importance of the data that is involved. A software developer can think in terms of graphs and graph algorithms without attaching concrete meaning to vertices and edges.

Notes

Working out an automated solution to a problem is thus a learning experience for both software developers and their clients. Software developers are learning the domain that the clients work in. They are also learning the values of the client: what form of data presentation is most useful to the client, what kinds of data are crucial and require special protective measures.

The clients are learning to see the range of possible solutions that software technology can provide. They are also learning to evaluate the possible solutions with regard to their effectiveness in meeting the clients needs.

If the problem to be solved is complex then it is not reasonable to assume that the best solution will be worked out in a short period of time. The clients do, however, want a timely solution. In most cases, they are not willing to wait until the perfect solution is worked out. They want a reasonable solution soon; perfection can come later. To develop a timely solution, software developers need to know the requirements: how the software should behave. The principle of anticipation of change recognizes the complexity of the learning process for both software developers and their clients. Preliminary requirements need to be worked out early, but it should be possible to make changes in the requirements as learning progresses.

Coupling is a major obstacle to change. If two components are strongly coupled then it is likely that changing one will not work without changing the other.

Cohesiveness has a positive effect on ease of change.

Did u know? **Is cohesive components are easier to reuse?**

Yes, cohesive component are easier to reuse when requirements change. If a component has several tasks rolled up into one package, it is likely that it will need to be split up when changes are made.

Generality

The principle of generality is closely related to the principle of anticipation of change. It is important in designing software that is free from unnatural restrictions and limitations. One excellent example of an unnatural restriction or limitation is the use of two digit year numbers, which has led to the "year 2000" problem: software that will garble record keeping at the turn of the century. Although the two-digit limitation appeared reasonable at the time, good software frequently survives beyond its expected lifetime.

For another example where the principle of generality applies, consider a customer who is converting business practices into automated software. They are often trying to satisfy general needs, but they understand and present their needs in terms of their current practices. As they become more familiar with the possibilities of automated solutions, they begin seeing what they need, rather than what they are currently doing to satisfy those needs. This distinction is similar to the distinction in the principle of abstraction, but its effects are felt earlier in the software development process.

Incremental Development

Fowler and Scott give a brief, but thoughtful, description of an incremental software development process. In this process, you build the software in small increments; for example, adding one use case at a time.

An incremental software development process simplifies verification. If you develop software by adding small increments of functionality then, for verification, you only need to deal with

Notes

the added portion. If there are any errors detected then they are already partly isolated so they are much easier to correct.

A carefully planned incremental development process can also ease the handling of changes in requirements. To do this, the planning must identify use cases that are most likely to be changed and put them towards the end of the development process.

Consistency

The principle of consistency is recognition of the fact that it is easier to do things in a familiar context. For example, coding style is a consistent manner of laying out code text. This serves two purposes. First, it makes reading the code easier. Second, it allows programmers to automate part of the skills required in code entry, freeing the programmer's mind to deal with more important issues.

At a higher level, consistency involves the development of idioms for dealing with common programming problems.

Consistency serves two purposes in designing graphical user interfaces. First, a consistent look and feel makes it easier for users to learn to use software. Once the basic elements of dealing with an interface are learned, they do not have to be relearned for a different software application. Second, a consistent user interface promotes reuse of the interface components. Graphical user interface systems have a collection of frames, panes, and other view components that support the common look. They also have a collection of controllers for responding to user input, supporting the common feel. Often, both look and feel are combined, as in pop-up menus and buttons. These components can be used by any program.

Meyer applies the principle of consistency to object-oriented class libraries. As the available libraries grow more and more complex it is essential that they be designed to present a consistent interface to the client.

Example: Most data collection structures support adding new data items. It is much easier to learn to use the collections if the name add is always used for this kind of operation.

Self Assessment

Fill in the blanks:

1. A data structure component is often easier to use if these two concerns are divided as much as possible into separate sets of…….
2. After the software is design to meet other criteria, it's …….. can be checked and analysed to see where the time is being spent.
3. A software developer can think in terms of graphs and graph algorithms without attaching concrete meaning to ……. and edges.
4. It is important in …….. software that is free from unnatural restrictions and limitations.
5. The principle of …….. is recognition of the fact that it is easier to do things in a familiar context.
6. Graphical user interface systems have a collection of frames, …….. , and other view components that support the common look.
7. A carefully planned incremental development process can also ease the handling of changes in……...

Notes

7.3 Planning Practices

The planning activity encompasses a set of management and technical practices that enable the software team to define a road maps it travels toward its strategic goal and tactical objectives. There are many different planning philosophies. Regardless of the rigor with which planning is conducted, the following principles always apply :

- Understand the scope of the project.
- Involve the customer in the planning activity.
- Recognize that planning is iterative.
- Estimate based on what you know.
- Consider risk as you define the plan.
- Be realistic.
- Adjust granularity as you define the plan.
- Define how you intend to ensure quality.
- Describe how you intend to accommodate change.
- Track the plan frequently and make adjustments as required.

Task What questions must be asked and answered to develop a realistic project plan:

1. Why is the system being developed?
2. What will be done?
3. When will it be accomplished?
4. Who is responsible for a function?
5. Where are they organizationally located?
6. How will the job be done technically and managerially?
7. How much of each resource is needed?

7.4 Modeling Practices

Model-Based Software Engineering (MBSE) is the idea of achieving code reuse and perform maintenance and product development through the use of software modeling technology and by splitting the production of software into two parallel engineering processes namely domain engineering and application engineering.

It presents the specific activities and responsibilities that are required of engineers who use the model-based verification paradigm and describes proposed approaches for integrating model-based verification into an organization's software engineering practices. The approaches outlined are preliminary concepts for the integration of model building and analysis techniques into software engineering review and inspection practices. These techniques are presented as both practices within peer review processes and as autonomous engineering investigations. The objective of this topic is to provide a starting point for the use of model-based verification techniques and a framework for their evaluation in real-world applications. It is expected that

Notes

the results of pilot studies that employ the preliminary approaches described here will form the basis for improving the practices themselves and software verification generally.

7.4.1 Analysis Modeling Principles

Analysis models represent customer requirements. Design models provide a concrete specification for the construction of the software. In analysis models, software is depicted in three domains. These domains are information domain, functional domain and behavioral domain.

Analysis modeling focuses on three attributes of software: information to be processed, function to be delivered and behavior to be exhibited. There are set of principles which relate analysis methods:

- The data that flows in and out of the system and data stores collectively are called information domain. This information domain should be well understood and represented.
- The functions of the software effect the control over internal and external elements. These functions need to be well defined.
- The software is influenced with external environment. Software behaves in a certain manner. This behavior should be well defined.
- Partitioning is a key strategy in analysis modeling. Divide the models depicting information, function and behavior in a manner which uncovers detail in hierarchical way.
- Description of problem from end-user's perspective is the start point of analysis modeling. Task should move from essential information toward implementation detail.

7.4.2 Design Modeling Principles

Design models provide a concrete specification for the construction of the software. It represents the characteristics of the software that help the practitioners to construct it effectively. Design modeling represents the dummy model of the thing that is to be built. In software systems, the design model provides different views of the system.

A set of design principles when applied creates a design that exhibits both internal and external quality factors. A high quality design can be achieved.

- The work of analysis model is to describe the information domain of problem, user functions, analysis classes with methods. The work of design model is to translate information from analysis model into an architecture. The design model should be traceable to analysis model.
- A data design simplifies program flow, design and implementation of software components easier. It is as important as designing of processing functions.
- In design modeling, the interfaces should be designed properly. It will make integration much easier and increase efficiency.
- Design should always start considering the architecture of the system that is to be built.
- End user plays an important role in developing a software system. User Interface is the visible reflection of the software. User Interface should be in terms of the end-user.
- Component functionality should focus on one and only one function or sub-function.

Notes

- In design modeling, the coupling among components should be as low as is needed and reasonable.
- Design models should be able to give information developers, testers and people who will maintain the software. In other words, it should be easily understandable.
- Design models should be iterative in nature. With each iteration, the design should move towards simplicity.

Self Assessment

Fill in the blanks:

8. Analysis models represent …….requirements.
9. The data that flows in and out of the system and data stores collectively are called …….. domain.
10. A data design simplifies program flow, design and ……. of software components easier.
11. Analysis modeling focuses on three attributes of software: information to be processed, function to be delivered and ……. to be exhibited.

7.5 Construction Practice

In software engineering practice, construction practice includes coding and testing tasks and principles. Testing is a process of finding out an error. A good testing technique is one which gives the maximum probability of finding out an error. The main objective of testers is to design tests that uncover different classes of errors with minimum amount of effort and time.

Some testing principles include:

- From a customer point of view, severe defects are one that cause failure of the program so all the tests that are defined should meet the requirements of the customer.
- Planning of tests should be done long before testing begins. When the design model is made, test cases cab be defined at that very point. Tests can be planned before any code has been generated.
- Generally, individual components are tested first and then the focus is shifted to integrated components and finally the entire system.
- It is not possible to execute different combination of paths during testing. This means that exhaustive testing is not possible.

In a broader software design context, we begin "in the large" by focusing on software architecture and end "in the small" focusing on components. In testing, the focus is reversed.

7.5.1 Coding Principle

Coding principles helps you to maintain good coding practice along with concrete product development. Coding principles also helps you write excellent quality of code with huge difference to overall performance and scalability of your application. Following are the most important Coding Principles which can save a lot for you when you are writing quality application for you or your client. Ensuring these coding principles you can save development time, management time and conquer lots of other bottlenecks which generally arise in later

Notes

development phases. You can also use the idea of PHP Design patterns which can speed up the development process by providing tested, proven development paradigms.

Don't Repeat Yourself (DRY)

The concept here is that anything you use in your code should have a single, unambiguous representation. This means if you are storing a number, only store it in a single place (don't have PI stored in three different places). Not only is multiple storage wasteful, but it multiplies your effort if you want to change, especially if you have to go hunting through several files to find where the numbers are defined. Multiple representations are also a great way to generate bugs if you forget to change some of them.

Caution This also applies to code; don't repeat chunks of code that do the same thing – have a single version and put it in a function.

Test as you Write

As much as possible, test your code as you write it. These tests will often be quick and easy ones, such as checking that the value of pi you're using is really what it should be, but if you perform these little checks while you're working on (and thinking about) that piece of code, you'll save yourself a lot more effort having to come back later and fix bugs. You'll find that you can perform a lot of simple tests very quickly as you go along; once you're in the habit, you really don't spend a lot of time doing it. But the time you save yourself later on can be considerable. As well as informal testing, using a test harness and writing automated tests will help guarantee code you have already written keeps doing what you expect and helps prevent bugs you have already fixed from reappearing.

Reduce Dependencies as much as Possible

A dependency is a connection between two chunks of code, for example, using the same variable, and the more of these your code has, the more things you have to keep track of. When writing and debugging a chunk of code, the last thing you want is arbitrary other chunks being able to interact with it, because it'll very quickly become unmanageable. Much better would be for each chunk of code to be highly de-coupled from the rest of the code, with only very specific connections to the rest of the program. This keeps things much simpler and easy to manage. In practice, this means things like compartmentalising your code into chunks (which you were doing anyway, right?), avoiding global variables (with the possible exception of explicitly fixed constants such as PI) and the like. If you're feeling keen, writing object oriented code is excellent for this, as this has encapsulation built into the process.

Validate your Data

At some point, someone will feed garbage into your carefully crafted code. In fact, part of your testing should be feed garbage input into your code to check that it recognises it! If your code is validating the data it is given then it should be able to deal intelligently with this, even if "intelligently" means "crash but tell the user what has gone wrong and why". Assertions are an ideal way to make the program stop as soon as something is wrong. They work by 'asserting' that something is true and if it isn't then the program stops.

Notes

Example: Assert (x >= 0)

If the variable x is less than zero then the program will immediately stop at this point.

During development this kind of information is invaluable and since the program has stopped at the first point it spotted something wrong, you don't have to work backwards from a garbage final output to find where the program failed.

Handle Errors Nicely

Asserts are a great way of validating data and are very useful during development, however once a program is in the hands of the users you want your error handling to be a little nicer than stopping the program immediately. There is nothing more frustrating than a program that just dies without warning or explanation. Most modern languages have support for handling problems your code encounters using Exceptions. Exceptions are generated when something goes wrong and bubble up until they are caught and dealt with. The advantage of exceptions is that they can be used without your code having to pass around error-code results from function to function. Using exceptions properly is a complex subject, because the exception handling represents a different path through the code, but a simple rule of thumb is: 'Throw an exception when a problem is first encountered, catch it at the first point that can deal with it'. In programs with a GUI (Graphical User Interface) this usually means there is a catchall at the user interface layer that displays a message to the user (or something similar) before attempting to save the data and then shutting down.

Keep it Simple

The simpler your code is, the easier it is to construct and maintain. So, subject to the constraints of our objectives, the simpler you can make your code the better. This has a connection to premature optimisation (see this post), because optimised code tends to be less simple. We think a reasonable rule-of-thumb is that unless most simple way of doing something will be obviously too inefficient, it's worth trying. You can always change it later and, because it's simple code, this shouldn't be too hard. One way to describe this is to paraphrase Albert Einstein: Code should be as simple as possible, but no simpler.

Tidy up after Yourself

Once you start to leave one or two things unfixed, it becomes much easier to leave "just one more", and soon your code is a wreck. There should not be a single "broken window" in the code you're building (the phrase "broken window" comes from a study that showed that a major indicator of the state of a building was a single broken window; once one is broken, people care a lot less about maintaining the rest, it seems). The book "The Pragmatic Programmer" has a nice description of this.

Learn New Tools

If all you have is a hammer, then all your problems tend to look like nails. The way to avoid this is to have more than one tool. In general, you want to have a good, broad selection of tools with which to write your code. A good way to acquire this is to try to learn the occasional new tool as you go along.

Notes

Notes These can be useful pieces of software, new techniques or whatever; the important thing is that it gives you at least one more good option for writing your code.

Maintain your Flow

Flow is a psychological state that you get into when you're absorbed in your work (sportsmen call it being "in the zone"). Have you ever gotten really into your work, so much so that suddenly an hour has passed without you noticing? That's flow! In this state, you tend to be very productive and very focused on the task at hand. Therefore, you want to try to stay in this state for as much of the time as possible. It can take as much as 15 minutes to get into the flow, so it's important to minimise things that will pull you out of the flow (once you're out, that's 15 minutes largely wasted). That means try to avoid distractions, turn off the e-mail alert beep, listen to music to block out background noise, turn off your web browser. Whatever it takes!

Make your Code Unsurprising

When someone glances at a chunk of code, they can often form a quick impression of what that chunk of code does. It is convenient when this impression is accurate; it can be a real problem if the impression is misleading and they makes changes to the code before they realise. The 'principle of least surprise' is that you should try to make your code's actual functionality as close as possible to the typical quick impression. Or, to put it another way, you should try to write your code so that it communicates its functionality accurately in a very short (pain-free) amount of time. This means doing things like picking informative variable/function names, writing informative (and succinct) comments, and making the layout easy to read.

Don't Program by Coincidence

'Programming by coincidence' is what happens when you "just get started coding" without thinking about what it is you actually want to achieve. You write some code and it seems to work. Great. So you write some more. That also seems to works. So you keep on doing this until one day you add some code and your software falls over. And you have no idea why. Think of this as the coding equivalent of a random walk; sure, you're moving, but are you going to end up in the right place? To avoid this, realise that a bit of planning and careful thought (including as the project progresses) is a very good thing.

Code Entropy

Disordered code is bad, because it's more likely to contain bugs. Higher entropy is more disordered which is bad. So you should try to keep the entropy of your code as low as possible. This means taking care to keep things neat and tidy as you write the code. It also means fixing bugs and refactoring messy bits of code. One important aspect of code entropy is that any time you make a change to your code, the level of entropy will tend to increase (unless you're very careful; fixing a bug, for example). If you're constantly rewriting your code, you'll be introducing new bugs all the time and invalidating the testing you've done on that code.

Task "So bear in mind that a bit of stability in your code is a good thing." Analyze this statement.

Notes

7.5.2 Coding Concepts

Computer programs are collections of instructions that tell a computer how to interact with the user, interact with the computer hardware and process data. The first programmable computers required the programmers to write explicit instructions to directly manipulate the hardware of the computer. This "machine language" was very tedious to write by hand since even simple tasks such as printing some output on the screen require 10 or 20 machine language commands. Machine language is often referred to as a "low level language" since the code directly manipulates the hardware of the computer.

By contrast, higher level languages such as C, C++, Pascal, Cobol, Fortran, ADA and Java are called "compiled languages". In a compiled language, the programmer writes more general instructions and a compiler (a special piece of software) automatically translates these high level instructions into machine language. The machine language is then executed by the computer. A large portion of software in use today is programmed in this fashion.

We can contrast compiled programming languages with interpreted programming languages. In an interpreted programming language, the statements that the programmer writes are interpreted as the program is running. This means they are translated into machine language on the fly and then execute as the program is running. Some popular interpreted languages include Basic, Visual Basic, Perl and shell scripting languages such as those found in the UNIX environment.

We can make another comparison between two different models of programming. In structured programming, blocks of programming statements (code) are executed one after another. Control statements change which blocks of code are executed next.

In object oriented programming, data are contained in objects and are accessed using special methods (blocks of code) specific to the type of object. There is no single "flow" of the program as objects can freely interact with one another by passing messages.

In this tutorial, we focus only on structured programming.

Program Structure

Virtually all structured programs share a similar overall structure:

- Statements to establish the start of the program
- Variable declaration
- Program statements (blocks of code)

The following is a simple example of a program written in several different programming languages. We call this the "Hello World" example since all the program does is print "Hello World" on the computer screen.

Notes The Perl and Basic languages are technically not compiled languages. These language statements are "interpreted" as the program is running.

Notes

Language	Example program

"C"	#include <stdio.h>
	void main() {
	printf("Hello World");
	}

C++	#include <iostream>
	
	int main()
	{
	cout << "Hello World";
	return 0;
	}

Pascal	program helloworld (output);
	begin
	writeln('Hello World');
	end.

Oracle PL/SQL	CREATE OR REPLACE PROCEDURE helloworld AS
	BEGIN
	DBMS_OUTPUT.PUT_LINE('Hello World');
	END;

Java	class helloworld
	{
	public static void main (String args [])
	{
	System.out.println ("Hello World");
	}
	}

Perl	#!/usr/local/bin/perl -w print "Hello World";
Basic	print "Hello World"

7.5.3 Testing Principles

Whether you have performed testing for a number of years or have just started, this is a course you really need to take. You will be exposed to a number of testing practices; some of which will likely be new to you. Drawing on his over 30 years of experience with all aspects of the software process, Dr. Hanna will use real life examples to illustrate every point. The course will introduce you to a Scenario-Based test design methodology. The course also covers the different types of testing performed at each phase of the software lifecycle and the issues involved in these types of tests, clarifies testing terminology, and much more.

Principles of Good Testing

- ***Complete testing isn't possible:*** No matter how much you test, it is impossible to achieve total confidence. The only exhaustive test is one that leaves the tester exhausted!
- ***Test work is creative and difficult:*** You must understand and probe what the system is supposed to do. You must understand and stress the limitations and constraints. You must understand the domain and application in depth.
- ***Testing is risk-based:*** We can't identify all risks of failure. Risk assessments indicate how much to test and what to focus on.

Notes

- ***Analysis, planning, and design are important:*** Test objectives must be identified and understood. Tests must be planned and designed systematically. Without a road map, you will get lost.
- ***Motivation is important:*** You cannot be effective if you don't care about the job. You must want to find problems and enjoy trying to break the system.
- ***Time and resources are important:*** You can't be effective if you don't have the time or resources to do the job.
- ***Timing of test preparation matters a lot:*** Early test preparation leads to an understanding of project requirements and design. Early test preparation uncovers and prevents problems. Early tests improve the effectiveness of subsequent reviews and inspections.
- ***Measuring and tracking coverage is essential:*** You need to know what requirements, design, and code have and have not been covered. Complex software is too difficult to cover without systematic measurement.

Self Assessment

Fill in the blanks:

12. The main objective of testers is to design tests that uncover different classes of …….. with minimum amount of effort and time.
13. By contrast, higher level languages such as C, C++, Pascal, Cobol, Fortran, ADA and Java are called "…….. languages".
14. …….. indicate how much to test and what to focus on.
15. Complex software is too difficult to cover without …….. measurement.

7.6 Summary

- Separation of concerns is recognition of the need for human beings to work within a limited context. The human mind is limited to dealing with approximately seven units of data at a time.
- When specifying the behavior of a data structure component, there are often two concerns that need to be dealt with: basic functionality and support for data integrity.
- A data structure component is often easier to use if these two concerns are divided as much as possible into separate sets of client functions.
- The planning activity encompasses a set of management and technical practices that enable the software team to define a road maps it travels toward its strategic goal and tactical objectives.
- The objective of this topic is to provide a starting point for the use of model-based verification techniques and a framework for their evaluation in real-world applications.
- In software engineering practice, construction practice includes coding and testing tasks and principles. Testing is a process of finding out an error.
- A good testing technique is one which gives the maximum probability of finding out an error.

Notes

7.7 Keywords

MBSE: Model-Based Software Engineering

SEMS: Software Engineering Maturity Spectrum

7.8 Review Questions

1. Separation of concerns is recognition of the need for human beings to work within a limited context. Explain.
2. The principle of abstraction is another specialization of the principle of separation of concerns. Comment.
3. Model-Based Software Engineering (MBSE) is the idea of achieving code reuse and perform maintenance. Analyze.
4. Analysis models represent customer requirements. Give Examples.
5. Explain why description of problem from end-user's perspective is the start point of analysis modeling.
6. A good testing technique is one which gives the maximum probability of finding out an error. Analyze.
7. Coding principles helps you to maintain good coding practice along with concrete product development. Comment.
8. The advantage of exceptions is that they can be used without your code having to pass around error-code results from function to function. Examine.
9. The 'principle of least surprise' is that you should try to make your code's actual functionality as close as possible to the typical quick impression. Discuss.
10. In object oriented programming, data are contained in objects and are accessed using special methods (blocks of code) specific to the type of object. Explain with reasons.

Answers: Self Assessment

1. client functions
2. run time
3. vertices
4. designing
5. consistency
6. panes
7. requirements
8. customer
9. information
10. implementation
11. behavior
12. errors
13. compiled
14. Risk assessments
15. systematic

7.9 Further Readings

Notes

Books

Rajib Mall, *Fundamentals of Software Engineering*, Second Edition, PHI.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering a Practioner's Approach*, 5th edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering*, Sixth edition, Pearson Education.

Online links

http://ksi.cpsc.ucalgary.ca/KAW/KAW96/herlea/FINAL.html

http://en.wikipedia.org/wiki/Requirements_analysis

Notes

Unit 8: Design Engineering

CONTENTS

Objectives

Introduction

8.1 What is Software Design?

8.2 Design Process

8.3 Design Quality

8.4 Design Concepts

8.4.1 Modularity

8.5 Summary

8.6 Keywords

8.7 Review Questions

8.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize the design process and design quality
- Scan the various design concepts

Introduction

Software design is a process of problem solving and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view.

8.1 What is a Software Design?

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design.

Software design can be considered as putting solution to the problem(s) in hand using the available capabilities. Hence the main difference between Software analysis and design is that the output of the analysis of a software problem will be smaller problems to solve and it should not deviate so much even if it is conducted by different team members or even by entirely different groups. But since design depends on the capabilities, we can have different designs for the same problem depending on the capabilities of the environment that will host the solution (whether it is some OS, web , mobile or even the new cloud computing paradigm). The solution will depend also on the used development environment (Whether you build a solution from scratch or using reliable frameworks or at least implement some suitable design patterns)

Notes

Did u know? **What is Platform-Independent Model (PIM)?**

A Platform-Independent Model (PIM) in software engineering is a model of a software system or business system that is independent of the specific technological platform used to implement it.

8.2 Design Process

The phrase software design is often used to characterize the discipline that is also called software engineering—the discipline concerned with the construction of software that is efficient, reliable, robust, and easy to maintain.

Interface Design

Bringing Design to Software implies that the object of design is software, leaving out considerations of the interface devices that are the inevitable embodiment of software for the user. Design cannot be neatly divided into compartments for software and for devices: The possibilities for software are both created and constrained by the physical interfaces. In today's world of computer applications, the vast majority of applications present themselves to users in a standard way—a visual display with a keyboard and mouse. But the future of computing will bring richer resources to physical human–computer interactions. Some new devices are already in use on a modest scale.

Example: Pen-based personal digital assistants (PDAs), virtual-reality goggles and gloves, and computers embedded in electromechanical devices of all kinds.

Researchers are exploring further possibilities, including tactile input and output devices, immersive environments, audio spaces, wearable computers, and a host of gadgets that bear little resemblance to today's personal computer or workstation. As experience with a wider variety of devices accumulates, the design of interaction based on new combinations of devices and software will be an important emerging topic in what we have—for the moment—called software design.

Human-computer Interaction

Whenever someone designs software that interacts with people, the effects of the design extend beyond the software itself to include the experiences that people will have in encountering and using that software. A person encountering any artifact applies knowledge and understanding, based on wide variety of cognitive mechanisms grounded in human capacities for perception, memory, and action. Researchers in human-computer interaction have studied the mental worlds of computer users, developing approaches and methods for predicting properties of the interactions and for supporting the design of interfaces. Although it would be overstating the case to say that the cognitive analysis of human-computer interaction has led to commonly accepted and widely applied methods, there is a substantial literature that can be of value to anyone designing interactive software.

Art

The experience of a person who is interacting with a computer system is not limited to the cognitive aspects that have been explored in the mainstream literature on human-computer

Notes

interaction. As humans, we experience the world in aesthetic, affective, and emotional terms as well. Because computing evolved initially for use in the laboratory and the office, non-cognitive aspects have been largely ignored, except by creators of computer games. Yet, whenever people experience a piece of software—whether it be a spreadsheet or a physics simulation—they have natural human responses. They experience beauty, satisfaction, and fun, or the corresponding opposites.

As computing becomes integrated into technologies for entertainment, and as the typical user moves from the well-regimented office to the home recreation room, software designers will need to focus more on the affective dimensions of human response. We can learn from the history of other human communication media, and can adapt the principles and techniques of novelists, film makers, composers, visual artists, and many other designers in what are loosely called the arts. Designing for the full range of human experience may well be the theme for the next generation of discourse about software design.

Perhaps even more difficult than the task of defining software is the task of defining design. A dictionary provides several loosely overlapping meanings, and a glance at the design section in a library or bookstore confuses the issue even more. Although we label it with a noun, design is not a thing. The questions that we can ask fruitfully are about the activity of designing.

Design is Conscious

People may refer to an object as being well designed whenever it is well suited to its environment, even if this suitability resulted from a process of unintentional evolution. In this book, we concentrate on what happens when a designer reflects on and brings focus to the design activity. Consciousness about designing does not imply the application of a formal, consistent, or comprehensive theory of design or of a universal methodology. Systematic principles and methods at times may be applicable to the process of design, but there is no effective equivalent to the rationalized generative theories applied in mathematics and traditional engineering. Design consciousness is still pervaded by intuition, tacit knowledge, and gut reaction.

Design keeps Human Concerns in the Center

All engineering and design activities call for the management of tradeoffs. Real-world problems rarely have a correct solution of the kind that would be suitable for a mathematics problem or for a textbook exercise. The designer looks for creative solutions in a space of alternatives that is shaped by competing values and resource needs. In classical engineering disciplines, the tradeoffs can often be quantified: material strength, construction costs, rate of wear, and the like. In design disciplines, the tradeoffs are more difficult to identify and to measure. The designer stands with one foot in the technology and one foot in the domain of human concerns, and these two worlds are not easily commensurable.

As an example, it is easy for software designers to fall into a single-minded quest, in which ease of use (especially for beginning users) becomes a holy grail. But what is ease of use? How much does it matter to whom? A violin is extremely difficult for novices to play, but it would be foolhardy to argue that it should therefore be replaced by the autoharp. The value of an artifact may lie in high-performance use by virtuosos, or in ease of use for some special class of users, such as children or people with disabilities. There is room for the software equivalents of high-strung racing cars alongside automatic-transmission minivans.

Design is a Dialog with Materials

The ongoing process of designing is iterative at two levels: iteration by the designer as a piece of current work develops, and iteration by the community as successive generations reveal new

possibilities for the medium. Designing is not a process of careful planning and execution, but rather it is like a dialog, in which the dialog partner—the designed object itself—can generate unexpected interruptions and contributions. The designer listens to the emerging design, as well as shapes it.

Design always implies a medium of construction, and new technologies bring with them new domains for design. Architecture as we know it appeared with the technology for building with stone. Graphic design emerged with the technologies of printing, and modern product design flourished with the development of plastics that expanded the possible variety of forms for everyday products.

Task The computer has produced the field of interaction design, in which new kinds of virtualities can be created and modified with a velocity that is unprecedented. What do you understand form this statement?

Design is Creative

It is one thing to lay out a list of criteria for good design. It is quite another thing to do design well. Designer lacks the comforting restraints of a well-organized engineering discipline, because designing is inherently messy; it includes, but goes beyond, the ability to be creative in solving problems. It begins with creativity in finding the problems—envisioning the needs that people have but do not recognize.

Design is more an art than a science—it is spontaneous, unpredictable, and hard to define. The skill of the artist–designer is not reducible to a set of methods, and is not learned through the kind of structured curriculum that serves in science and engineering. On the other hand, it is not a mysterious gift. There is a long and rich tradition of education in the design fields; it draws on the interaction between learner and teacher, designer and critic.

Design is Communication

Previous sections have described the interaction between the user and his world, and the interaction between the designer and her materials. What matters most is the interaction between these two interactions. A virtuality is neither just what the designer thinks it is, nor what any particular user thinks it is. It exists in the ongoing dialog between designers and users. To succeed in designing, we need to understand how designers convey meaning.

At a surface level, designed artifacts communicate content. Skilled designers in every discipline know how to manage these layers of meaning. In addition, an active artifact—whether it be a computer program or a coffee maker—communicates to users about its use. A device as apparently simple as a door communicates to its users through convention. A door with a flat plate near shoulder level says "Push me!" One with a round knob says "Twist here." Although these messages are constrained by the physical mechanisms, they are also a matter of convention and learning, as every tourist finds out in trying to deal with everyday objects in an unfamiliar culture.

As is true of any human language, what is visible in a statement is only a small part of the full picture. A hearer of the sentence "I'm going to kill you!" cannot interpret the intended meaning without knowing the situational context—was it said to a friend walking onto the tennis court, in an anonymous note mailed to the President, or by a parent to a child who has just left dirty clothing on the bathroom floor again? The literal meaning is but the shadow of the meaning in context.

Notes

The same is true for the artifacts that we build, including software: People do not approach them in isolation. Every object appears in a context of expectations that is generated by the history of previous objects and experiences, and by the surroundings in the periphery—the physical, social, and historical context in which the object is encountered.

Design has Social Consequences

Much of the discussion on software design uses examples from generic mass-distribution software, such as word processors, operating systems, spreadsheets, graphics programs, and games. Although many key concerns of software design are addressed in these applications, others are not. These highly visible applications are part of a larger picture that includes a vast number of vertical applications (for example, a medical-office billing application) and systems designed for a specific workplace setting. In these more targeted applications, the designer can see and take into account the specific effects of the design on the people who will inhabit it.

Organizational aspects of software design in which we build integrated computer systems for specific organizations and workplaces. The needs and concerns of the people in those workplaces can lead to complex—and, at times, controversial—design considerations, which we can address only by making the social and political dimensions an explicit part of the analysis and of the design dialog. The designer takes on a role that includes organizational as well as technical design, and can enlist workers directly in this process through techniques such as participatory design.

Design is a Social Activity

In concentrating on the activity of an individual designer, we can fall into the error of assuming that the overall quality of a design is primarily a result of the qualities and activities of the creative individual. As Kelley points out, the designer operates in a larger setting, which is both facilitated and constrained by interactions with other people.

Self Assessment

Fill in the blanks:

1. Software design is a process of problem solving and …… for a software solution.
2. Designing software is an exercise in managing…….
3. Design cannot be neatly divided into compartments for …….. and for devices.
4. Design always implies a medium of construction, and new technologies bring with them new …….. for design.
5. …… teams also tend to be fluid, likewise often changing in the middle of the design process.

8.3 Design Quality

Software design principles represent a set of guidelines that helps us to avoid having a bad design. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices". According to Robert Martin there are three important characteristics of a bad design that should be avoided:

- ***Rigidity:*** It is hard to change because every change affects too many other parts of the system.

Notes

- ***Fragility:*** When you make a change, unexpected parts of the system break.
- ***Immobility:*** It is hard to reuse in another application because it cannot be disentangled from the current application.

Open Close Principle

Software entities like classes, modules and functions should be open for extension but closed for modifications.

OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don't have to change the class but to extend it. The same principle can be applied for modules, packages, libraries. If you have a library containing a set of classes there are many reasons for which you'll prefer to extend it without changing the code that was already written (backward compatibility, regression testing). This is why we have to make sure our modules follow Open Closed Principle.

When referring to the classes Open Close Principle can be ensured by use of Abstract Classes and concrete classes for implementing their behavior. This will enforce having Concrete Classes extending Abstract Classes instead of changing them. Some particular cases of this are Template Pattern and Strategy Pattern.

Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Further more it inverts the dependency: instead of writing our abstractions based on details, we should write the details based on abstractions.

Dependency Inversion or Inversion of Control are better know terms referring to the way in which the dependencies are realized. In the classical way when a software module (class, framework) need some other module, it initializes and holds a direct reference to it. This will make the 2 modules tight coupled. In order to decouple them the first module will provide a hook (a property, parameter) and an external module controlling the dependencies will inject the reference to the second one.

By applying the Dependency Inversion the modules can be easily changed by other modules just changing the dependency module.

Did u know? **What is Inversion of Control Container?**

Factories and Abstract Factories can be used as dependency frameworks, but there are specialized frameworks for that, known as Inversion of Control Container.

Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they don't use.

This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not

Notes

be there the classes implementing the interface will have to implement those methods as well. For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?

As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. We should avoid them.

Single Responsibility Principle

A class should have only one reason to change.

In this context a responsibility is considered to be one reason to change. This principle states that if we have two reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Single Responsibility Principle was introduced Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

Liskov's Substitution Principle

Derived types must be completely substitutable for their base types.

This principle is just an extension of the Open Close Principle in terms of behavior meaning that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.

Liskov's Substitution Principle was introduced by Barbara Liskov in a 1987 Conference on Object Oriented Programming Systems Languages and Applications, in Data abstraction and hierarchy.

Self Assessment

Fill in the blanks:

6. Abstractions should not depend on ……...
7. Derived types must be completely …… for their base types.

8.4 Design Concepts

In order to deal with the complexity, software is designed in layers. When a programmer is worrying about the detailed design of one module, there are probably hundreds of other modules and thousands of other details that he cannot possibly worry about at the same time. For example, there are important aspects of software design that do not fall cleanly into the categories of data structures and algorithms. Ideally, programmers should not have to worry about these other aspects of a design when designing code.

This is not how it works, however, and the reasons start to make sense. The software design is not complete until it has been coded and tested. Testing is a fundamental part of the design validation and refinement process. The high level structural design is not a complete software design; it is just a structural framework for the detailed design. We have very limited capabilities

Notes

for rigorously validating a high level design. The detailed design will ultimately influence (or should be allowed to influence) the high level design at least as much as other factors. Refining all the aspects of a design is a process that should be happening throughout the design cycle. If any aspect of the design is frozen out of the refinement process, it is hardly surprising that the final design will be poor or even unworkable.

On any software project of typical size, problems like these are guaranteed to come up. Despite all attempts to prevent it, important details will be overlooked. This is the difference between craft and engineering. Experience can lead us in the right direction. This is craft. Experience will only take us so far into uncharted territory. Then we must take what we started with and make it better through a controlled process of refinement. This is engineering.

As just a small point, all programmers know that writing the software design documents after the code instead of before, produces much more accurate documents. The reason is now obvious. Only the final design, as reflected in code, is the only one refined during the build/test cycle. The probability of the initial design being unchanged during this cycle is inversely related to the number of modules and number of programmers on a project.

Caution It rapidly becomes indistinguishable from zero.

In software engineering, we desperately need good design at all levels. In particular, we need good top level design. The better the early design, the easier detailed design will be. Designers should use anything that helps. Structure charts, Booch diagrams, state tables, PDL, etc. — if it helps, then use it. We must keep in mind, however, that these tools and notations are not a software design. Eventually, we have to create the real software design, and it will be in some programming language. Therefore, we should not be afraid to code our designs as we derive them. We simply must be willing to refine them as necessary.

There is as yet no design notation equally suited for use in both top level design and detailed design. Ultimately, the design will end up coded in some programming language. This means that top level design notations have to be translated into the target programming language before detailed design can begin. This translation step takes time and introduces errors. Rather than translate from a notation that may not map cleanly into the programming language of choice, programmers often go back to the requirements and redo the top level design, coding it as they go. This, too, is part of the reality of software development.

It is probably better to let the original designers write the original code, rather than have someone else translate a language independent design later. What we need is a unified design notation suitable for all levels of design. In other words, we need a programming language that is also suitable for capturing high level design concepts. This is where C++ comes in. C++ is a programming language suitable for real world projects that is also a more expressive software design language. C++ allows us to directly express high level information about design components. This makes it easier also helps the process of detecting design errors. This results in a more robust design, in essence a better engineered design.

Ultimately, a software design must be represented in some programming language, and then validated and refined via a build/test cycle. Any pretense otherwise is just silliness. Consider what software development tools and techniques have gained popularity. Structured programming was considered a breakthrough in its time. Pascal popularized it and in turn became popular. Object oriented design is the new rage and C++ is at the heart of it. Now think about what has not worked. CASE tools? Popular, yes; universal, no. Structure charts? Same thing. Likewise, Warner-Orr diagrams, Booch diagrams, object diagrams, you name it. Each has its strengths, and a single fundamental weakness — it really isn't a software design. In fact the

Notes

only software design notation that can be called widespread is PDL, and what does that look like.

This says that the collective subconscious of the software industry instinctively knows that improvements in programming techniques and real world programming languages in particular are overwhelmingly more important than anything else in the software business. It also says that programmers are interested in design. When more expressive programming languages become available, software developers will adopt them.

Also consider how the process of software development is changing. Once upon a time we had the waterfall process. Now we talk of spiral development and rapid prototyping. While such techniques are often justified with terms like "risk abatement" and "shortened product delivery times", they are really just excuses to start coding earlier in the life cycle. This is good. This allows the build/test cycle to start validating and refining the design earlier.

Notes It also means that it is more likely that the software designers that developed the top level design are still around to do the detailed design.

Engineering is more about how you do the process than it is about what the final product looks like. We in the software business are close to being engineers, but we need a couple of perceptual changes. Programming and the build/test cycle are central to the process of engineering software. We need to manage them as such. The economics of the build/test cycle, plus the fact that a software system can represent practically anything, makes it very unlikely that we will find any general purpose methods for validating a software design. We can improve this process, but we can not escape it.

One final point: the goal of any engineering design project is the production of some documentation. Obviously, the actual design documents are the most important, but they are not the only ones that must be produced. Someone is eventually expected to use the software. It is also likely that the system will have to be modified and enhanced at a later time. This means that auxiliary documentation is as important for a software project as it is for a hardware project. Ignoring for now users manuals, installation guides, and other documents not directly associated with the design process, there are still two important needs that must be solved with auxiliary design documents.

The first use of auxiliary documentation is to capture important information from the problem space that did not make it directly into the design. Software design involves inventing software concepts to model concepts in a problem space. This process requires developing an understanding of the problem space concepts. Usually this understanding will include information that does not directly end up being modeled in the software space, but which nevertheless helped the designer determine what the essential concepts were, and how best to model them. This information should be captured somewhere in case the model needs to be changed at a later time.

The second important need for auxiliary documentation is to document those aspects of the design that are difficult to extract directly from the design itself. These can include both high level and low level aspects. Many of these aspects are best depicted graphically. This makes them hard to include as comments in the source code. This is not an argument for a graphical software design notation instead of a programming language. This is no different from the need for textual descriptions to accompany the graphical design documents of hardware disciplines. Never forget that the source code determines what the actual design really is, not the auxiliary documentation. Ideally, software tools would be available that post processed a source code design and generated the auxiliary documentation. That may be too much to expect. The next

Notes

best thing might be some tools that let programmers (or technical writers) extract specific information from the source code that can then be documented in some other way. Undoubtedly, keeping such documentation up to date manually is difficult. This is another argument for the need for more expressive programming languages. It is also an argument for keeping such auxiliary documentation to a minimum and keeping it as informal as possible until as late in the project as possible. Again, we could use some better tools, otherwise we end up falling back on pencil, paper, and chalk boards.

The design of software is often depicted by graphs that show components and their relationships. For example, a structure chart shows the calling relationships among components. Object-oriented design is based on various graphs, as well. Such graphs are abstractions of the software, devised to depict certain design decisions. Coupling and cohesion are attributes that summarizes the degree of interdependence or connectivity among subsystems and within subsystems, respectively. When used in conjunction with measures of other attributes, coupling and cohesion can contribute to an assessment or prediction of software quality.

(a) ***Abstraction:*** When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.

(b) ***Architecture:*** An architecture is not a simple flat view of the component topology, though an architecture diagram showing the components and relationships among them is a central thinking and communicating tool for the architects and the development team, and others they partner with. Our architecture needs to include:

- *Meta-architecture:* the architectural vision, style, principles, key communication and control mechanisms, and concepts that guide the team of architects in the creation of the architecture.
- *Architectural views:* Just as building architectures are best envisioned in terms of a number of complementary views or models, so too are software architectures. In particular, structural views help document and communicate the architecture in terms of the components and their relationships, and are useful in assessing architectural qualities like extensibility. Behavioral views are useful in thinking through how the components interact to accomplish their assigned responsibilities and evaluating the impact of what-if scenarios on the architecture. Behavioral models are especially useful in assessing run-time qualities such as performance and security. Execution views help in evaluating physical distribution options and documenting and communicating decisions.

(c) ***Design Patters:*** A software design pattern describes a family of solutions to a software design problem. It consists of one or several software design elements such as modules, interfaces, classes, objects, methods, functions, processes, threads, etc., relationships among the elements, and a behavioral description. Example design patterns are Model/View/Controller, Blackboard, Client/Server, and Process Control. The purpose of design patterns is to capture software design know-how and make it reusable. Design patterns can improve the structure of software, simplify maintenance, and help avoid architectural drift. Design patterns also improve communication among software developers and empower less experienced personnel to produce high-quality designs.

(d) ***Modularity:*** The basic idea underlying modular design is to organize a complex system (such as a large program, an electronic circuit, or a mechanical device) as a set of distinct components that can be developed independently and then plugged together. Although this may appear a simple idea, experience shows that the effectiveness of the technique depends critically on the manner in which systems are divided into components and the

Notes

mechanisms used to plug components together. The following design principles are particularly relevant to parallel programming.

(e) ***Provide simple interfaces:*** Simple interfaces reduce the number of interactions that must be considered when verifying that a system performs its intended function. Simple interfaces also make it easier to reuse components in different circumstances. Reuse is a major cost saver. Not only does it reduce time spent in coding, design, and testing, but it also allows development costs to be amortized over many projects. Numerous studies have shown that reusing software is by far the most effective technique for reducing software development costs. As an example, a modular implementation of a climate modeling system may define distinct modules concerned with atmosphere modeling, ocean modeling, etc. The interfaces to each module can comprise a small set of procedures that access boundary data, advance the simulation, and so on. Hence, there is no need for the user to become familiar with the implementation of the various modules, which collectively may comprise hundreds of procedures and tens of thousands of lines of code.

(f) ***Information Hiding:*** Ensure that modules hide information. The benefits of modularity do not follow automatically from the act of subdividing a program. The way in which a program is decomposed can make an enormous difference to how easily the program can be implemented and modified. Experience shows that each module should encapsulate information that is not available to the rest of a program. This information hiding reduces the cost of subsequent design changes. For example, a module may encapsulate

- related functions that can benefit from a common implementation or that are used in many parts of a system,
- functionality that is likely to change during later design or deployment,
- aspects of a problem that are particularly complex, and/or
- code that is expected to be reused in other programs.

Notes Notice that we do not say that a module should contain functions that are logically related because, for example, they solve the same part of a problem. This sort of decomposition does not normally facilitate maintenance or promote code reuse.

(g) ***Use appropriate tools:*** While modular designs can in principle be implemented in any programming language, implementation is easier if the language supports information hiding by permitting the encapsulation of code and data structures. Fundamental mechanisms in this regard include the procedure (or subroutine or function) with its locally scoped variables and argument list, used to encapsulate code; the user-defined datatype, used to encapsulate data; and dynamic memory allocation, which allows subprograms to acquire storage without the involvement of the calling program. These features are supported by most modern languages (e.g., C++ , Fortran 90, and Ada) but are lacking or rudimentary in some older languages (e.g., Fortran 77).

(h) ***Design checklist:*** The following design checklist can be used to evaluate the success of a modular design. As usual, each question should be answered in the affirmative.

- Does the design identify clearly defined modules?
- Does each module have a clearly defined purpose? (Can you summarize it in one sentence?)

Notes

- Is each module's interface sufficiently abstract that you do not need to think about its implementation in order to understand it? Does it hide its implementation details from other modules?
- Have you subdivided modules as far as usefully possible?
- Have you verified that different modules do not replicate functionality?
- Have you isolated those aspects of the design that are most hardware specific, complex, or otherwise likely to change?

8.4.1 Modularity

Modularity refers to the division of software into separate modules which are differently named and addressed and are integrated later on in order to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to large number of reference variables, control paths, global variables, etc. The desirable properties of a modular system are:

- Each module is a well defined system that can be used with other applications.
- Each module has a single specific purpose.
- Modules can be separately compiled and stored in a library.
- Modules can use other modules.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Modularity thus enhances the clarity of design which in turn eases coding, testing, debugging, documenting and maintenance of the software product. It might seem to you that on dividing a problem into sub problems indefinitely, the effort required to develop it becomes negligibly small. However, the fact is that on dividing the program into numerous small modules, the effort associated with the integration of these modules grows. Thus, there is a number N of modules that result in the minimum development cost. However, there is no defined way to predict the value of this N.

In order to define a proper size of the modules, we need to define an effective design method to develop a modular system. Following are some of the criteria defined by Meyer for the same:

- ***Modular decomposability:*** The overall complexity of the program will get reduced if the design provides a systematic mechanism to decompose the problem into sub-problems and will also lead to an efficient modular design.
- ***Modular composability:*** If a design method involves using the existing design components while creating a new system it will lead to a solution that does not reinvent the wheel.
- ***Modular understandability:*** If a module can be understood as a separate standalone unit without referring to other modules it will be easier to build and edit.
- ***Modular continuity:*** If a change made in one module does not require changing all the modules involved in the system, the impact of change-induced side effects gets minimized.
- ***Modular protection:*** If an unusual event occurs affecting a module and it does not affect other modules, the impact of error-induced side effects will be minimized.

Notes

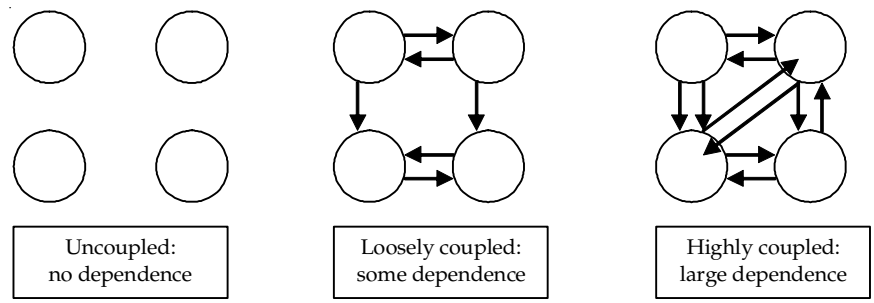
Modular Design

A modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of different parts of a system. We discuss the various concepts of modular design in detail in this section:

1. ***Functional Independence:*** This concept evolves directly from the concepts of abstraction, information hiding and modularity. Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules. Independence is important because it makes implementation easier and faster. The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well. Thus, functional independence is a good design feature which ensures software quality. It is measured using two criteria: coupling and cohesion.

2. ***Module Coupling:***

Figure 8.1: Module Coupling



Task Analyze at what extend the functional dependence affects the modular design.

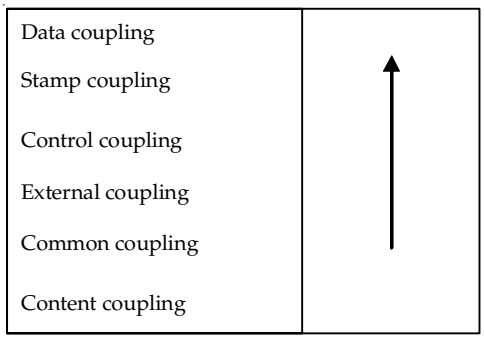
Coupling is the measure of degree of interdependence amongst modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. Uncoupled modules have no interdependence at all within them. The various types of coupling techniques are depicted in Figure 8.1.

A good design is the one that has low coupling. Coupling is measured by the number of interconnections between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors. Different types of coupling are content, common, external, control, stamp and data. The level of coupling in each of these types is given in Figure 8.2.

The direction of the arrow in Figure 8.2 points from the lowest coupling to highest coupling. The strength of coupling is influenced by the complexity of the modules, the type of connection and the type of communication. Modifications done in the data of one block may require changes in other block of a different module which is coupled to the former module.

Notes

Figure 8.2: Types of Modules Coupling



Notes However, if the communication takes place in the form of parameters then the internal details of the modules are no required to be modified while making changes in the related module.

Given two procedures X and Y, the type of coupling can be identified in them.

(a) *Data coupling:* When X and Y communicates by passing parameters to one another and not unnecessary data. Thus, if a procedure needs a part of a data structure, it should be passed just that and not the complete thing.

(b) *Stamp Coupling:* Although X and Y make use of the same data type but perform different operations on them.

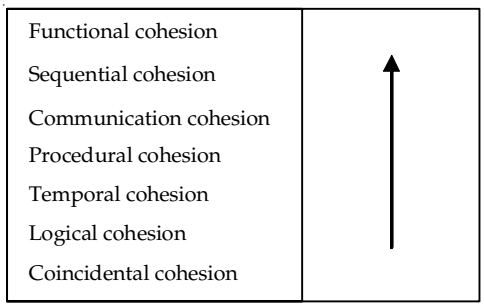
(c) *Control Coupling (activating):* X transfers control to Y through procedure calls.

(d) *Common Coupling:* Both X and Y use some shared data e.g. global variables. This is the most undesirable, because if we wish to change the shared data, all the procedures accessing this shared data will need to be modified.

(e) *Content Coupling:* When X modifies Y either by branching in the middle of Y or by changing the local data values or instructions of Y.

3. ***Cohesion:*** Cohesion is the measure of the degree of functional dependence of modules. A strongly cohesive module implements functionality that interacts little with the other modules. Thus, in order to achieve higher interaction amongst modules a higher cohesion is desired. Different types of cohesion are listed in Figure 8.3.

Figure 8.3: Types of Module Cohesion



Notes

The direction of the arrow in Figure 8.3 indicates the worst degree of cohesion to the best degree of cohesion. The worst degree of cohesion, coincidental, exists in the modules that are not related to each other in any way. So, all the functions, processes and data exist in the same module.

Logical is the next higher level where several logically related functions or data are placed in the same module.

At times a module initializes a system or a set of variables. Such a module performs several functions in sequence, but these functions are related only by the timing involved. Such cohesion is said to be temporal.

When the functions are grouped in a module to ensure that they are performed in a particular order, the module is said to be procedurally cohesive. Alternatively, some functions that use the same data set and produce the same output are combined in one module. This is known as communicational cohesion.

If the output of one part of a module acts as an input to the next part, the module is said to have sequential cohesion. Because the module is in the process of construction, it is possible that it does not contain all the processes of a function. The most ideal of all module cohesion techniques is functional cohesion. Here, every processing element is important to the function and all the important functions are contained in the same module. A functionally cohesive function performs only one kind of function and only that kind of function.

Given a procedure carrying out operations A and B, we can identify various forms of cohesion between A and B:

(a) ***Functional Cohesion:*** A and B are part of one function and hence, are contained in the same procedure.

(b) ***Sequential Cohesion:*** A produces an output that is used as input to B. Thus they can be a part of the same procedure.

(c) ***Communicational Cohesion:*** A and B take the same input or generate the same output. They can be parts of different procedures.

(d) ***Procedural Cohesion:*** A and B are structured in the similar manner. Thus, it is not advisable to put them both in the same procedure.

(e) ***Temporal Cohesion:*** Both A and B are performed at moreover the same time. Thus, they cannot necessarily be put in the same procedure because they may or may not be performed at once.

(f) ***Logical Cohesion:*** A and B perform logically similar operations.

(g) ***Coincidental Cohesion:*** A and B are not conceptually related but share the same code.

Information Hiding: In computer science, information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, through an explicit exporting policy and through reliance on the short form as the primary vehicle for class documentation.

Notes

Functional Dependence: Designing modules in such a way that each module has specific functional requirements. Functional independence are measured using two terms cohesion and coupling.

Did u know? **What is the basic difference between the Coupling and Cohesion?**

The difference between Coupling and Cohesion is a distinction on a process of change, not a static analysis of code's quality.

A Coupling and Cohesion Measurement

Decomposition is one of the oldest and most fundamental techniques employed in software engineering. It results in a good system design which consists of relatively independent components which interact only in order to achieve system goals.

We can model a real-world problem by breaking it up into more manageable sub-problems until these are small enough to be represented accurately. The problem is then defined by the representations of the subproblems and the relationships between them. Similarly, our design and subsequent implementation will also consist of a number of related parts. Although the terms used to denote the parts (subsystems, modules, classes, components, . . .) and relationships have changed over the years the benefits of a modular system have endured. The basic idea is simple: put things that belong together in one place. This leads to independent 'pluggable' modules which are easy to develop, maintain and replace and which present simple interfaces to clients.

As always, there is a catch. There is no single correct way to perform decomposition and different choices can lead to different designs with very different properties. For example, techniques based on decomposition with respect to data flow tend to produce different results to those of techniques based on decomposition with respect to control.

A related problem is assessing the degree to which each module's contents 'belong together' and the relative independence of modules. Cohesion is a measure of internal module strength while coupling describes the strength of inter-module relationships. They are not independent coupling is lowered (and cohesion raised) when communicating activities are located in the same module. Cohesion and coupling are an ingredient in many design techniques.

The advent of object-oriented software development has brought many benefits but has added further complexity to the concepts of cohesion and coupling by increasing the number of available component and relationship types. For example, in conventional procedural languages such as Pascal or C the dominant relationship is procedure invokes procedure. In OO languages components include methods, classes and packages while relationships include association, inheritance and containment in addition to invocation. Only for very small systems is a subjective assessment of the levels of cohesion possible and sufficient. Measures which are more objective and amenable to automated collection are required for systems of realistic size. Cohesion is multi-faceted, involving complex interactions between many components, and it is not possible to represent it adequately with only scalar quantitative metrics without significant loss of information. Software visualization excels in problem domains having many variables, complex interactions and large data volumes.

Did u know? **What is the importance of software visualization in software engineering?**

Software visualization involves applying information visualization principles to the software engineering domain.

Notes

Measuring Cohesion

Module cohesion was defined by Yourdan and Constantine as how tightly bound or related its internal elements are to one another and an attribute that can be used to predict properties of implementations such as ease of debugging, ease of maintenance, and ease of modification. Since cohesion refers to the degree to which module components belong together, cohesion measurement should prove to be a very useful restructuring tool.

Following the original guidelines, skilled engineers conduct the assessment of module cohesion. These engineers would apply a set of subjective criteria to analyze associations between \processing elements and classify the nature of these associations. Because of the subjective nature of the assessment, the measurement of module cohesion has been difficult to automate, and cohesion has not been effectively used as a software quality indicator. Several approaches have been used to develop objective, automatable methods for measuring module cohesion. The first approach, an association-based approach, is to formalize the notion of the associations between processing elements as a set of rules concerning data dependencies in module code. This method requires the analysis of code-level information and thus cannot be applied before code is written. The second approach, a slice-based approach. They measure functional cohesion in terms of the connections between code data tokens on module output slices. This method also requires code level information. Cohesion measures for object-oriented software have also been defined using a slice-based approach, and by analyzing the connectivity between methods through common references to instance variables. Method bodies are needed to apply these code-level class cohesion measures.

Coupling of a subsystem characterizes its interdependence with other subsystems. A subsystem's cohesion, on the other hand, characterizes its internal interdependencies. When used in conjunction with other attributes, measurements of a subsystem's coupling and cohesion can contribute to software quality models.

Did u know? **How can be the abstraction of a software system can be represented?**

An abstraction of a software system can be represented by a graph and a module (subsystem) by a subgraph. Software-design graphs depict components and their relationships.

Measuring Coupling

Coupling is an internal software attribute that can be used to indicate the degree of interdependence among the components of a software system. It has been recognized that good software design should obey the principle of low coupling. A system that has strong coupling may make it more complex because it is difficult to understand, change, and correct highly interrelated components in the system. Coupling is therefore considered to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability.

A first-order principle of software architecture is to increase cohesion and reduce coupling. Cohesion (interdependency within module) strength/level names: (from worse to better, high cohesion is good).

- ***Refinement:*** Modern software development is a complicated process especially when a software system becomes large and complicated. Software developers must apply software refinement in order to proceed from a high-level abstract model to a final executable

software system by adding more details over time. Class diagrams are important because they represent the static structure of a software system, and therefore we design a UML profile extending the UML meta model to support class diagram refinement.

- ***Refactoring:*** Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring." [Martin Fowler] Refactoring is used to improve code quality, reliability, and maintainability throughout the software lifecycle. Code design and code quality are enhanced with refactoring. Refactoring also increases developer productivity and increases code reuse.
- ***Design Classes:*** Class diagrams are an aspect of UML that describe a static design of the objects, and their relationships to each other, in an application. During analysis, class diagrams may just be the names of objects and how they interact, but as the design develops the details of each class, including attributes and methods, are displayed in the diagram. A class is represented by a rectangle divided into three rows. The top row contains the name of the class. The middle row lists the attribute names, while the third row lists the method names for the class. There are two major relationships that can exist between classes: inheritance and association. Inheritance, also know as generalization, describes a superclass/subclass relationship. An empty arrow that points from the subclass to the superclass represents a generalization relationship.

Self Assessment

Fill in the blanks:

8. A software design must be represented in some programming language, and then validated and refined via a …… cycle.
9. Software design involves inventing software concepts to model concepts in a …….. space.
10. Behavioral models are especially useful in assessing ……. qualities such as performance and security.
11. Execution views help in evaluating physical distribution options and …….. and communicating decisions.
12. …… are important because they represent the static structure of a software system.
13. Measures which are more objective and amenable to ……. collection are required for systems of realistic size.
14. ……. is a measure of internal module strength while coupling describes the strength of inter-module relationships.
15. The protection involves providing a stable interface which protects the remainder of the program from the …….

Notes

Caselet

Design Software Vendors see Big Market in India

Design software vendors see a big market emerging in India as outsourcing of engineering services gains momentum. The outsourced engineering services is considered a next big thing for Indian vendors, who are seen buying design software and tools to build capabilities to tap the new found opportunity.

"The engineering services provider (ESP) category is an emerging market segment for us where we are seeing some good growth," said Mr Kamal Kumar, Managing director of Delmia India, a subsidiary of Dassault Systemes.

While the ESP market in India is definitely adding to the kitty of the product lifecycle management (PLM) solution vendors like Dassault, PTC, UGS among others, there are no estimates available for the design software market in the ESP category. The PLM solutions help companies manage entire lifecycle of a product from the concept and design phase through product planning, visualisation and marketing to the point where the product is recycled.

In India, industry verticals like automotive, aerospace and discrete manufacturing are leading the adoption of PLM solutions. "We are seeing demand from not only global firms setting up their captive design centres in India, but also from the major Indian IT vendors, who are building capabilities to tap the engineering services segment," said Mr Vivek Marawah, Country Marketing Manager, UGS India.

Companies like TCS, Infosys Technologies, Satyam Computer, Tata Technologies, Infotech Enterprises among others are seen building up capabilities in the engineering services space. "As the offshore volumes of engineering services increase, we see a comparable rise in the usage and adoption of these design tools," Mr Marawah said.

Mr Natarajan Viswanathan, Managing Director, PTC India said the usage of software design tools would be a prelude to the adoption of PLM solutions, which will eventually happen over the next year.

It is estimated that only $10-15 billion of the $750 billion engineering services market is presently outsourced and has the potential to reach $150-225 billion by 2020.

8.5 Summary

- Whenever objects are created for people to use, design is pervasive. Design may be done methodically or offhandedly, consciously or accidentally.
- The phrase software design is often used to characterize the discipline that is also called software engineering—the discipline concerned with the construction of software that is efficient, reliable, robust, and easy to maintain.
- Software design principles represent a set of guidelines that helps us to avoid having a bad design.
- The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices".
- In order to deal with the complexity, software is designed in layers. When a programmer is worrying about the detailed design of one module, there are probably hundreds of

Notes

other modules and thousands of other details that he cannot possibly worry about at the same time.

- A modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of different parts of a system.

8.6 Keywords

Coupling: Coupling is an internal software attribute that can be used to indicate the degree of interdependence among the components of a software system.

Dependency Inversion Principle: Dependency Inversion Principle states that we should decouple high level modules from low level modules.

Design Patterns: A software design pattern describes a family of solutions to a software design problem.

Modular Design: A modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of different parts of a system.

8.7 Review Questions

1. In software engineering why do we desperately need good design at all levels?
2. Explain with example that how the modular design reduces the design complexity?
3. "Software Design is more an art than a science." Analyze.
4. Clients should not be forced to depend upon interfaces that they don't use. Explain.
5. Discuss the set of guidelines provided by the software design principles that helps to avoid having a bad design.
6. Dependency Inversion and Inversion of Control have the same meaning? Why or why not? Explain briefly.
7. A good design is the one that has low coupling. Explain.
8. Engineering is more about how you do the process than it is about what the final product looks like. Discuss.
9. The design of software is often depicted by graphs that show components and their relationships. Comment.
10. "Two modules that are tightly coupled are strongly dependent on each other." Explain why?

Answers: Self Assessment

1.	planning	2.	complexity
3.	software	4.	domains
5.	Software development	6.	details
7.	substitutable	8.	build/test
9.	problem	10.	run-time
11.	documenting	12.	Class diagrams

Notes

13. Automated
14. Cohesion
15. implementation

8.8 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering*, Second Edition, PHI.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering a Practioner's Approach*, 5th edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering*, Sixth edition, Pearson Education.

Online links

http://en.wikipedia.org/wiki/Software_design

http://www.developerdotstar.com/mag/articles/reeves_design.html

Notes

Unit 9: System Engineering

CONTENTS

Objectives

Introduction

9.1 System Engineering Hierarchy

9.1.1 System Modeling

9.1.2 System Simulation

9.2 Types of Models

9.3 System Modeling

9.3.1 Hatley-Pirbhai Modeling

9.3.2 System Modeling with UML

9.4 Role of Modeling and Simulation

9.5 CODIS Framework

9.6 Summary

9.7 Keywords

9.8 Review Questions

9.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize the system engineering hierarchy
- Describe the system modeling and system simulation
- Examine Hatley-Pirbhai modeling
- Discuss system modeling with UML

Introduction

Systems Engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems Engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs. Systems Engineering is an interdisciplinary process that ensures that the customer's needs are satisfied throughout a system's entire life cycle. This process is comprised of the following seven tasks.

- ***State the problem:*** Stating the problem is the most important systems engineering task. It entails identifying customers, understanding customer needs, establishing the need for change, discovering requirements and defining system functions.
- ***Investigate alternatives:*** Alternatives are investigated and evaluated based on performance, cost and risk.

Notes

- ***Model the system:*** Running models clarifies requirements, reveals bottlenecks and fragmented activities, reduces cost and exposes duplication of efforts.
- ***Integrate:*** Integration means designing interfaces and bringing system elements together so they work as a whole. This requires extensive communication and coordination.
- ***Launch the system:*** Launching the system means running the system and producing outputs — making the system do what it was intended to do.
- ***Assess performance:*** Performance is assessed using evaluation criteria, technical performance measures and measures — measurement is the key. If you cannot measure it, you cannot control it. If you cannot control it, you cannot improve it.
- ***Re-evaluation:*** Re-evaluation should be a continual and iterative process with many parallel loops.

We live in a world of systems driven by cause and affect. Those systems include financial, production, inventory, biological, chemical, thermodynamic or workflow. Systems can be modeled as nodes representing system variables and connecting lines representing causal effects. The changing value of one variable can cause another to increase or decrease as described by equations. Understanding how a system really works is the first step towards using, improving, automating or explaining it to others.

This unit explores the use of modelling and simulation as a problem solving tool.

We undertake this discussion within the framework of a modelling and simulation project. This project framework embraces two key notions; first there is the notion of a 'system context'; that is, there is a system that has been identified for investigation, and second, there is a problem relating to the identified system that needs to be solved. Obtaining an acceptable solution to this problem is the goal of the modelling and simulation project. We use the term 'system' in its broadest possible sense; it could, for example, include the notions of a process or a phenomenon. Furthermore, physical existence of the system is not a prerequisite; the system in question may simply be a concept, idea, or proposal. What is a prerequisite, however, is the requirement that the system in question exhibit 'behaviour over time,' in other words, that it be a dynamic system.

Systems, or more specifically dynamic systems, are one of the most pervasive notions of our contemporary world. Broadly speaking, a dynamic system is a collection of interacting entities that produces some form of behaviour that can be observed over an interval of time. There are, for example, physical systems such as transportation systems, power generating systems, or manufacturing systems. On the other hand, in less tangible form, we have healthcare systems, social systems, and economic systems. Systems are inherently complex and tools such as modelling and simulation are needed to provide the means for gaining insight into features of their behaviour. Such insight may simply serve to provide the intellectual satisfaction of deeper understanding or, on the other hand, may be motivated by a variety of more practical and specific reasons such as providing a basis for decisions relating to the control, management, acquisition, or transformation of the system under investigation (the SUI).

The defining feature of the modelling and simulation approach is that it is founded on a computer-based experimental investigation that utilises an appropriate model for the SUI. The model is a representation or abstraction of the system. The use of models (in particular, mathematical models) as a basis for analysis and reasoning is well established in such disciplines as engineering and science. It is the emergence and widespread availability of computing power that has made possible the new dimension of experimentation with complex models and hence, the emergence of the modelling and simulation discipline.

Notes

It must be emphasised, furthermore, that there is an intimate connection between the model that is 'appropriate' for the study and the nature of the problem that is to be solved. The important corollary here is that there rarely exists a 'universal' model that will support all modelling and simulation projects that have a common system context. This is especially true when the system has some reasonable level of complexity. Consider, for example, the difference in the nature of a model for an airliner, first in the case where the model is intended for use in evaluating aerodynamic properties versus the case where it is simply a revenue-generating object within a business model. Identification of the most appropriate model for the project is possibly the most challenging aspect of the modelling and simulation approach to problem solving.

Although the word 'modelling' has a meaning that is reasonably confined in its general usage, the same cannot be said for the word 'simulation'. Nevertheless, the phrase 'modelling and simulation' does have a generally accepted meaning and implies two distinct activities. The modelling activity creates an object (i.e., a model) that is subsequently used as a vehicle for experimentation. This experimentation with the model is the simulation activity.

The word 'simulation' is frequently used alone in a variety of contexts. For example, it is sometimes used as a noun to imply a specialized computer program (as in, 'A simulation has been developed for the proposed system.'). It is also used frequently as an adjective (as in, 'The simulation results indicate that the risk of failure is minimal,' or 'Several extensions have been introduced into the language to increase its effectiveness for simulation programming'). These wide-ranging and essentially inconsistent usages of the word 'simulation' can cause regrettable confusion for neophytes to the discipline.

Caution As a rule, we avoid such multiplicity of uses of this word but, as will become apparent, we do use the word as an adjective in two specific contexts where the implication is particularly suggestive and natural.

9.1 System Engineering Hierarchy

Good system engineering begins with a clear understanding of context – the world view – and then progressively narrows focus until technical details are understood. System engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy.

9.1.1 System Modeling

System engineering process begins with a world of view which is refined to focus more fully on a specific domain of interest. Within a specific domain, the need for targeted system elements is analyzed. Finally, the analysis, design, and construction of targeted system element is initiated. Broad context is established at the top of the hierarchy and at the bottom, detailed technical activities are conducted. It is important for a system engineer narrows the focus of work as one moves downward in the hierarchy.

System modeling is an important element of system engineering process. System engineering model accomplishes the following:

- define processes.
- represent behavior of the process.
- define both exogenous and endogenous input to model.
- represent all linkages.

Notes

Some restraining factors that are considered to construct a system model are:

- Assumptions that reduce number of possible permutations and variations thus enabling a model to reflect the problem in a reasonable manner.
- Simplifications that enable the model to be created in a timely manner.
- Limitations that help to bound the system.
- Constraints that will guide the manner in which the model is created and the approach taken when the model is implemented.
- Preferences that indicate the preferred architecture for all data, functions , and technology.

The resultant system model may call for a completely automated or semi automated or a non automated solution.

Task System modeling is an important element of system engineering process. Analyze how?

9.1.2 System Simulation

After some consideration regarding a meaningful way of putting System, Model, and Simulation in an appropriate perspective I arrived at the following distinction.

System: A system exists and operates in time and space.

Model: A model is a simplified representation of a system at some particular point in time or space intended to promote understanding of the real system.

Simulation: A simulation is the manipulation of a model in such a way that it operates on time or space to compress it, thus enabling one to perceive the interactions that would not otherwise be apparent because of their separation in time or space.

Modeling and Simulation is a discipline for developing a level of understanding of the interaction of the parts of a system, and of the system as a whole. The level of understanding which may be developed via this discipline is seldom achievable via any other discipline.

A system is understood to be an entity which maintains its existence through the interaction of its parts. A model is a simplified representation of the actual system intended to promote understanding. Whether a model is a good model or not depends on the extent to which it promotes understanding. Since all models are simplifications of reality there is always a trade-off as to what level of detail is included in the model. If too little detail is included in the model one runs the risk of missing relevant interactions and the resultant model does not promote understanding. If too much detail is included in the model the model may become overly complicated and actually preclude the development of understanding. One simply cannot develop all models in the context of the entire universe, of course unless you name is Carl Sagan.

A simulation generally refers to a computerized version of the model which is run over time to study the implications of the defined interactions. Simulations are generally iterative in there development. One develops a model, simulates it, learns from the simulation, revises the model, and continues the iterations until an adequate level of understanding is developed.

Modeling and Simulation is a discipline, it is also very much an art form. One can learn about riding a bicycle from reading a book. To really learn to ride a bicycle one must become actively engaged with a bicycle. Modeling and Simulation follows much the same reality. You can learn

Notes

much about modeling and simulation from reading books and talking with other people. Skill and talent in developing models and performing simulations is only developed through the building of models and simulating them. It's very much a learning process as you go. From the interaction of the developer and the models emerges an understanding of what makes sense and what doesn't.

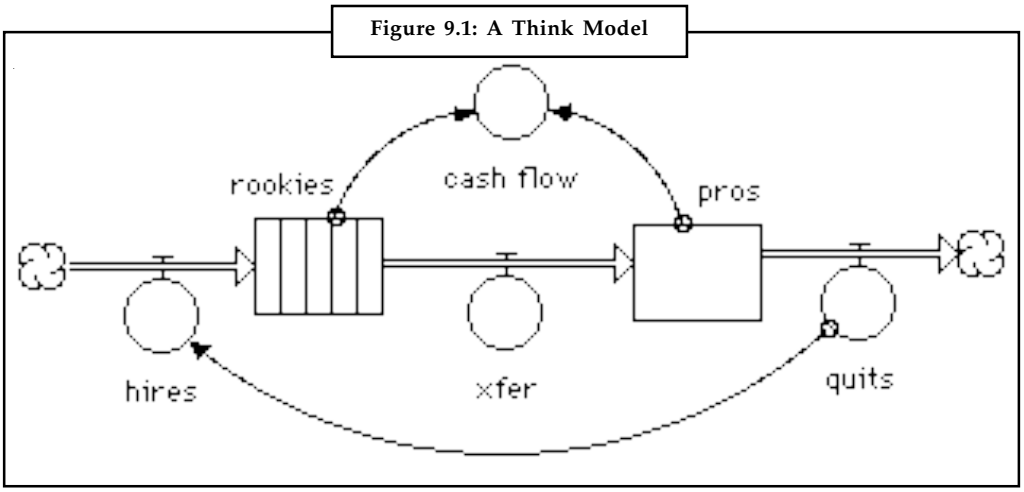
We repeatedly amazed at the ability of my models to point out my own ignorance. Through the activity of developing the model and then simulating it, the simulation says, "Based on your model and your set of assumptions, reality is absurd!" Often times the model is grossly incorrect, and other times the model produces great leaps in my understanding of how things actually work, and, quite often contrary to common sense. It is an amazingly wonderful journey my models and I are undertaking.

Detail complexity is associated with systems which have many component parts. Dynamic complexity is associated with systems which have cause and effect separated by time and or space. The understanding is that it is dynamic complexity that we have great difficulty dealing with because we are unable to readily see the connections between the parts of the system and their interactions. One of the great values of simulation is its ability to effect a time and space compression on the system, essentially allowing one to perceive, in a matter of minutes, interactions that would normally unfold over very lengthy time periods. This is probably best demonstrated by an example. The following example is an elaboration of one of the introductory models in the I think documentation from isee Systems.

Example: Consider a consulting company which has 120 employees. These 120 employees are composed of 60 rookies and 60 professionals. The company wishes to maintain the total number of employees at 120 so it hires a new rookie for each professional who quits. Rookies don't quit! Professionals quit at a rate of 10 per month and it takes 6 months to develop a professional from a rookie. Additionally, the company bills out rookies at $10k/month and professionals at $15k/month. All 120 employees are fully applied (I know it's a pipe dream).

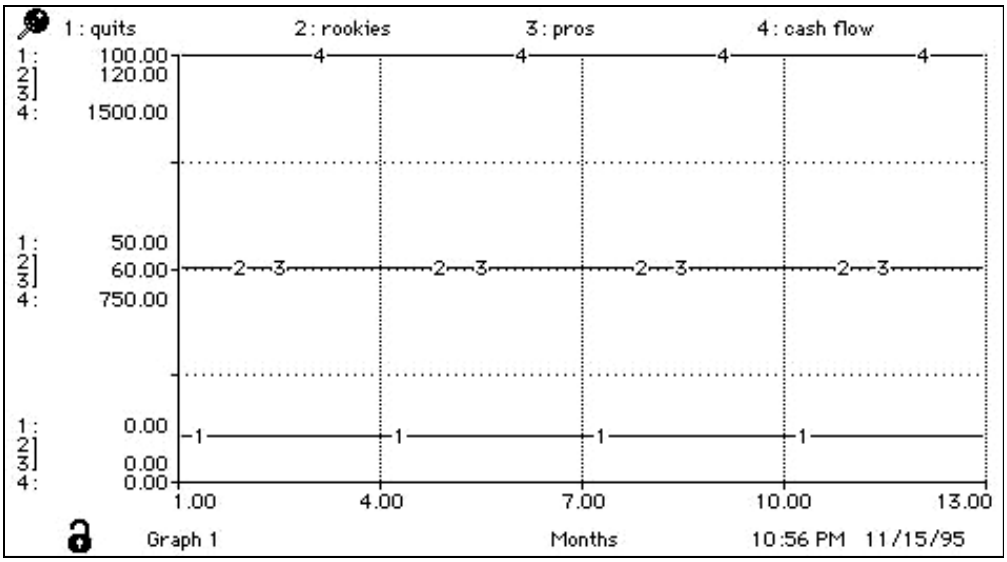
A think model for this system might look like the following:

Figure 9.1: A Think Model



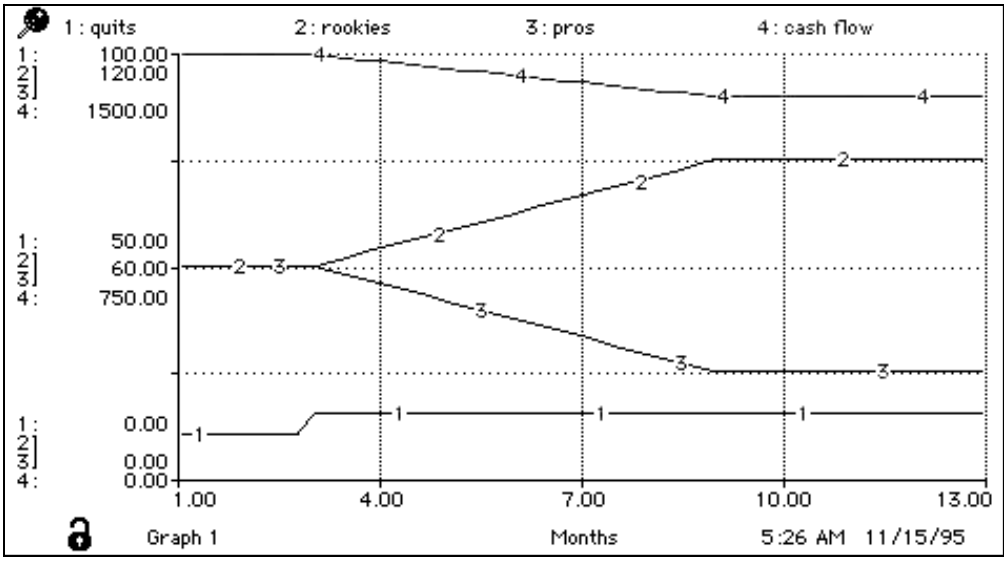
If you run this model you find it exists in essentially a steady state, and is about as exciting as watching paint dry!

Notes



Now, in the 10th month the company notices its revenue has dropped from $1.5m/month to $1.35m/month and it wonders what has happened. And where do you think it looks for the problem? All around the 10th month of course. And what does it find? The company finds that it still has 120 employees, yet there are now 30 professionals and 90 rookies. A most puzzling situation!

As it turns out, there was an organizational policy change made in month 3 which seemed to annoy professionals more than in the past, and the quit rate jumped from 10 to 15 professionals a month. The system, with it's built in hiring rule, essentially an auto pilot no thought action, hired one rookie for each professional that quit. What this one time transition in quit rate actually did was set off a 6 month transition within the organization leading to a new equilibrium state with 30 professionals and 90 rookies. The following graph represents this transition.



Thus, one of the real benefits of modeling and simulation is its ability to accomplish a time and space compression between the interrelationships within a system. This brings into view the results of interactions that would normally escape us because they are not closely related in time and space. Modeling and simulation can provide a way of understanding dynamic complexity.

Notes

Self Assessment

Fill in the blanks:

1. ……., or more specifically dynamic systems, are one of the most pervasive notions of our contemporary world.
2. The analysis, design, and …… of targeted system element are initiated.
3. A system is understood to be an entity which maintains its existence through the interaction of its……...

9.2 Types of Models

System Modelling

System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers

Different models present the system from different perspectives

- External perspective showing the system's context or environment
- Behavioural perspective showing the behaviour of the system
- Structural perspective showing the system or data architecture

Structured Methods

Structured methods incorporate system modelling as an inherent part of the method

Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models

CASE tools support system modelling as part of a structured method

Method Weaknesses

They do not model non-functional system requirements

They do not usually include information about whether a method is appropriate for a given problem

The may produce too much documentation

The system models are sometimes too detailed and difficult for users to understand

Model Types

Data processing model showing how the data is processed at different stages

Composition model showing how entities are composed of other entities

Architectural model showing principal sub-systems

Classification model showing how entities have common characteristics

Stimulus/response model showing the system's reaction to events

Notes

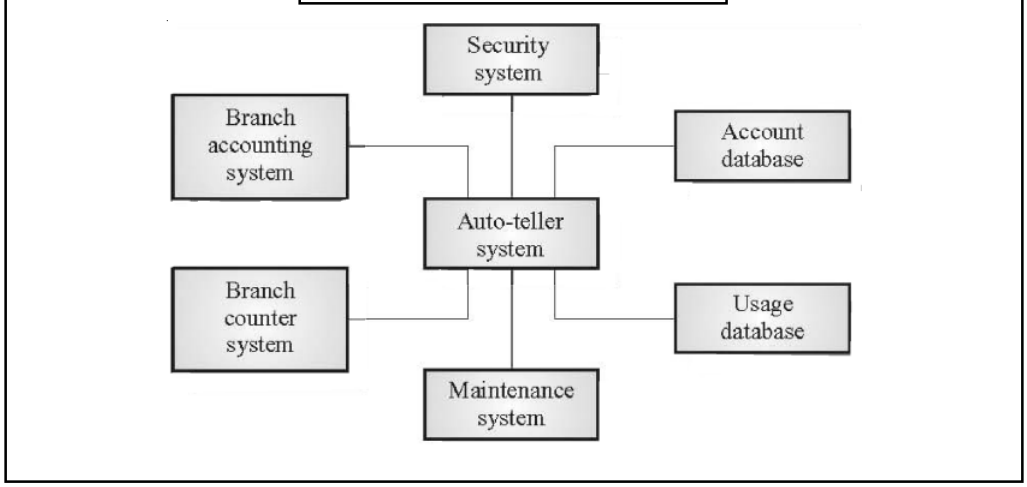
Context Models

Context models are used to illustrate the boundaries of a system.

Social and organisational concerns may affect the decision on where to position system boundaries

Architectural models show the a system and its relationship with other systems

Figure 9.2: Context of an ATM System

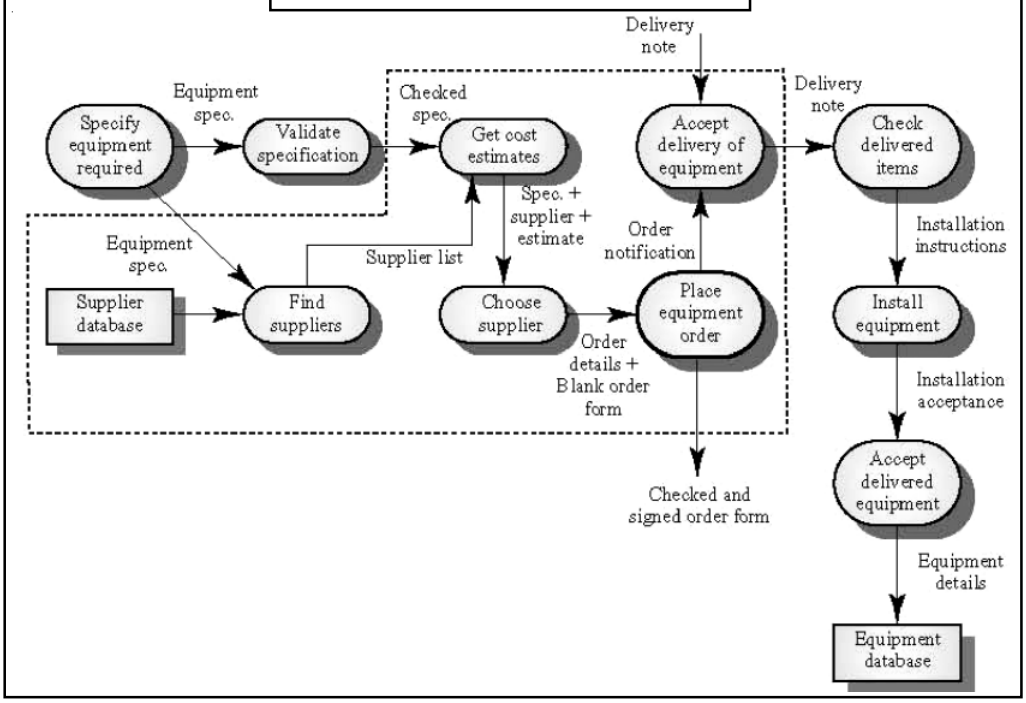


Process Models

Process models show the overall process and the processes that are supported by the system

Data flow models may be used to show the processes and the flow of information from one process to another

Figure 9.3: Equipment Procurement Process



Notes

Behavioral Models

Behavioral models are used to describe the overall behaviour of a system

Two types of behavioral model are shown here

- Data processing models that show how data is processed as it moves through the system
- State machine models that show the systems response to events

Did u know? **Explain the requirement of these models.**

Both of these models are required for a description of the system's behavior.

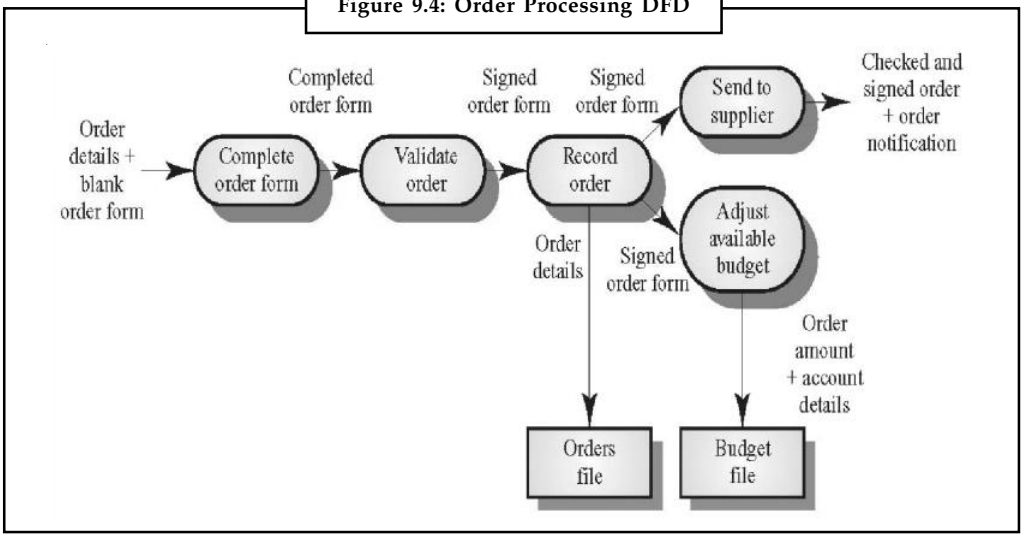
Data-processing Models

Data flow diagrams are used to model the system's data processing

These show the processing steps as data flows through a system

- Intrinsic part of many analysis methods
- Simple and intuitive notation that customers can understand
- Show end-to-end processing of data
- Order Processing DFD

Figure 9.4: Order Processing DFD



Data Flow Diagrams

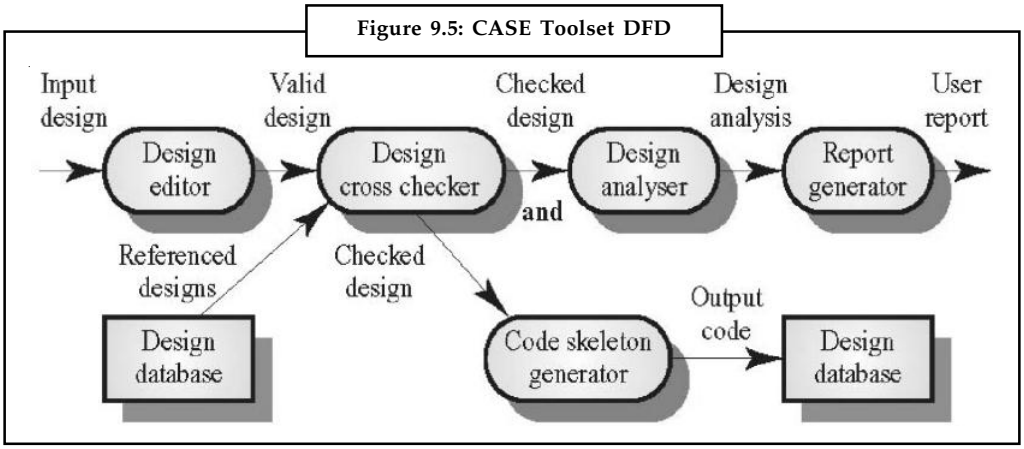
DFDs model the system from a functional perspective

Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system

Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment

Notes

Figure 9.5: CASE Toolset DFD



State Machine Models

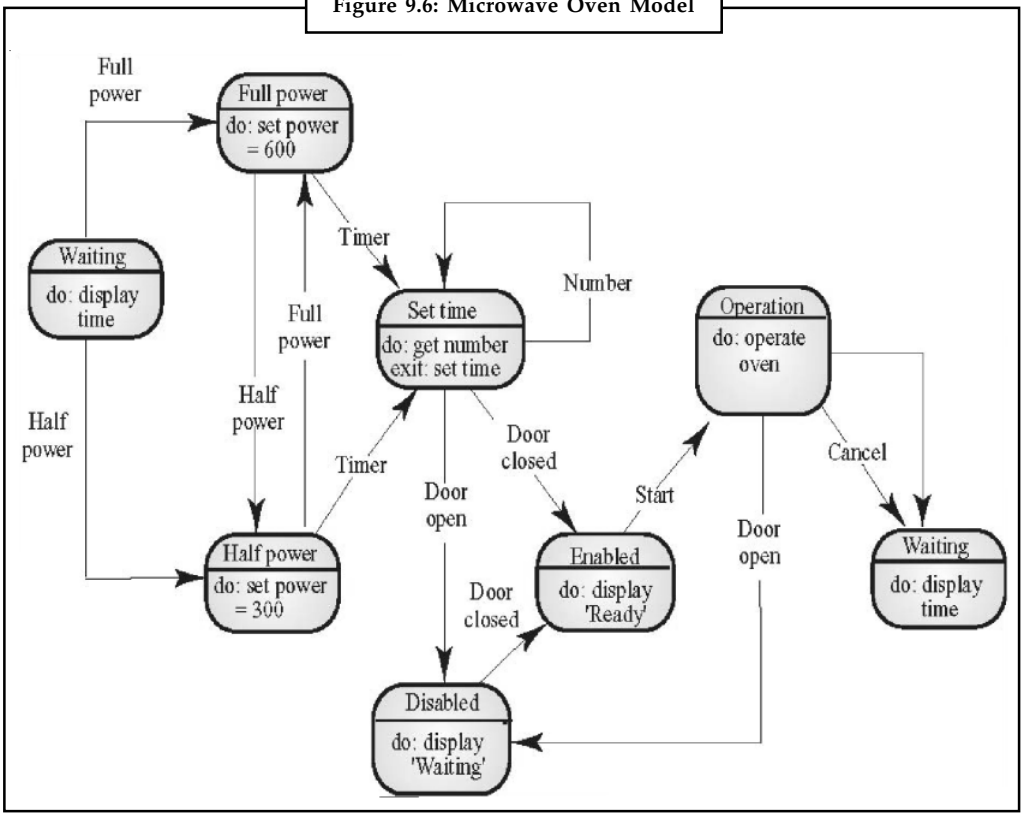
These model the behaviour of the system in response to external and internal events

They show the system's responses to stimuli so are often used for modelling real-time systems

State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another

State charts are an integral part of the UML

Figure 9.6: Microwave Oven Model



State Charts

Notes

Allow the decomposition of a model into sub-models (see following slide)

A brief description of the actions is included following the 'do' in each state

Can be complemented by tables describing the states and the stimuli

Figure 9.7: Microwave Oven Operation

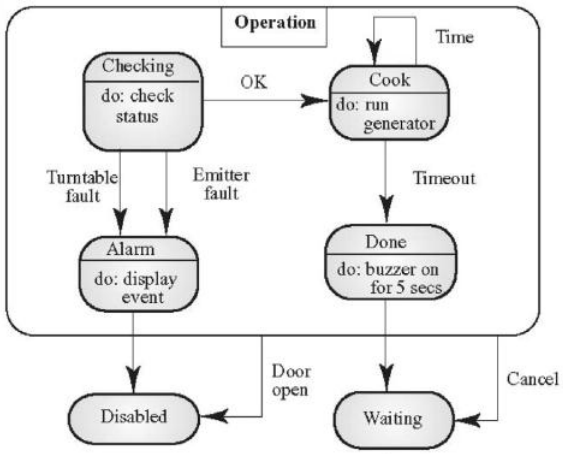
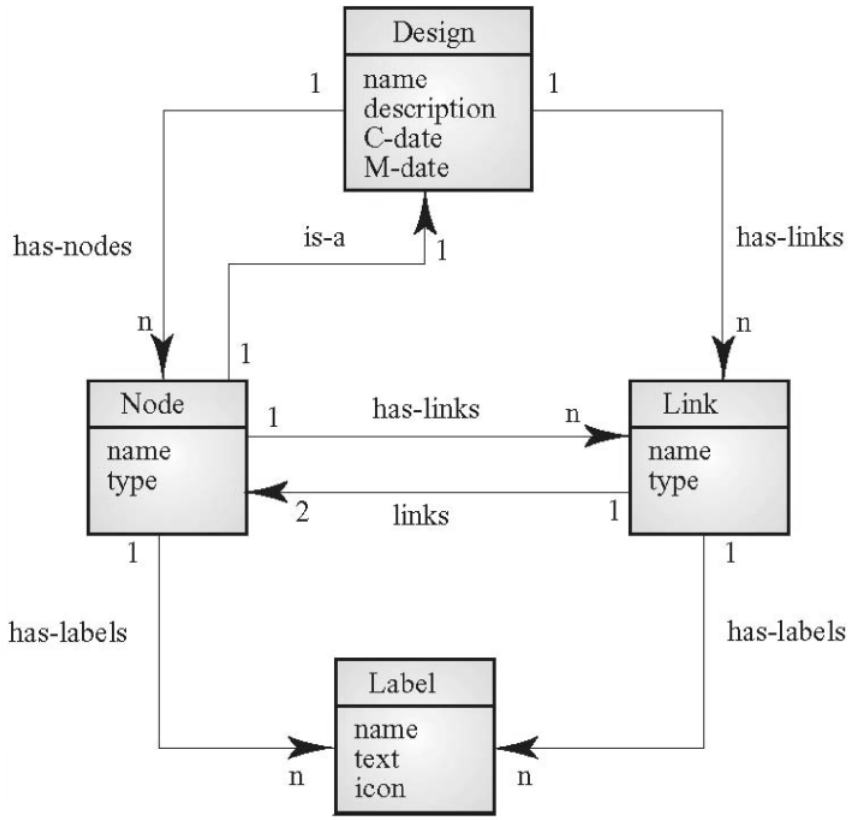


Figure 9.8: Software Design Semantic Model



Notes

Semantic Data Models

Used to describe the logical structure of data processed by the system

Entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes

Widely used in database design. Can readily be implemented using relational databases

No specific notation provided in the UML but objects and associations can be used

Data Dictionaries

Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included

Advantages

- Support name management and avoid duplication
- Store of organisational knowledge linking analysis, design and implementation

Many CASE workbenches support data dictionaries

Object Models

Object models describe the system in terms of object classes

An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object

Various object models may be produced

- Inheritance models
- Aggregation models
- Interaction models

Natural ways of reflecting the real-world entities manipulated by the system

More abstract entities are more difficult to model using this approach

Object class identification is recognised as a difficult process requiring a deep understanding of the application domain

Object classes reflecting domain entities are reusable across systems

Inheritance Models

Organise the domain object classes into a hierarchy

Classes at the top of the hierarchy reflect the common features of all classes

Object classes inherit their attributes and services from one or more super-classes. These may then be specialised as necessary

Class hierarchy design is a difficult process if duplication in different branches is to be avoided

Self Assessment

Notes

Fill in the blanks:

4. Classification model showing how entities have …… characteristics.
5. ……. models show the a system and its relationship with other systems.
6. …… are used to model the system's data processing.
7. No specific notation provided in the …….. but objects and associations can be used.

9.3 System Modeling

9.3.1 Hatley-Pirbhai Modeling

System models are hierarchical or layered as a system is represented at different levels of abstraction. The top level of hierarchy presents the complete system. The data objects, functions, behaviors are represented. As the hierarchy is refined or layered, component level detail is modeled and finally system models evolve into engineering models.

Hatley-Pirbhai modeling is an extension of the concept that every computer system can be modeled through the usage of an input-processing-output model by including the two additional features of user interface process and maintenance/self testing. These five components are added to a system model template to allow for modeling of the system which allows for proper assignment to the processing regions.

- The Hatley-Pirbhai model depicts input processing, and output along with the user interface and maintenance or self-test.
- It includes two more system features: user interface processing and maintenance and self-test processing.
- A system engineer can create a model of system components that sets a foundation for later steps.
- The system engineer allocates system elements to each of five processing regions within the template: user interface, input, system function and control, output, maintenance and self-test.
- A system context diagram (SCD) resides at the top level of hierarchy which defines all external producers of information used by the system, all external consumers of information created by the system and all entities that communicate through interface or perform maintenance and self-test.
- All the major subsystems are defined in a system flow diagram (SFD) which is derived from SCD.
- Information flows across the regions of system context diagram is used to guide the system engineer in developing system flow diagram.
- System flow diagram shows major subsystems and important lines of information.

Notes

9.3.2 System Modeling with UML

"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

The important point to note here is that UML is a 'language' for specifying and not a method or procedure. The UML is used to define a software system; to detail the artifacts in the system, to document and construct - it is the language that the blueprint is written in.

Notes The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process) – but in itself it does not specify that methodology or process.

UML defines the notation and semantics for the following domains:

- ***The User Interaction or Use Case Model:*** describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.
- ***The Interaction or Communication Model:*** describes how objects in the system will interact with each other to get work done.
- ***The State or Dynamic Model:*** State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows the system will implement.
- *The Logical or Class Model:* describes the classes and objects that will make up the system.
- ***The Physical Component Model:*** describes the software (and sometimes hardware components) that make up the system.
- ***The Physical Deployment Model:*** describes the physical architecture and the deployment of components on that hardware architecture.

The UML also defines extension mechanisms for extending the UML to meet specialized needs (for example Business Process Modeling extensions).

Self Assessment

Fill in the blanks:

8. A …… is a specification for behavior generation and the modeling process is concerned with the development of this specification.
9. …… is a constructive activity and this raises the natural question of whether the product (i.e., the model) is 'good enough.'
10. The fundamental requirement of any …….. is the replication of system behaviour within a physical environment that is as realistic as possible from the perspective of an operator.

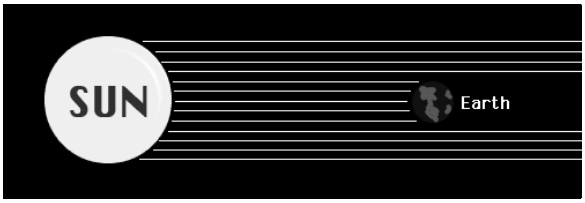
Notes

9.4 Role of Modeling and Simulation

Definitions of Systems and Models

A system is an assemblage of interrelated parts that work together by way of some driving process (see Figure 9.9). Systems are often visualized or modeled as component blocks that have connections drawn between them. For example, the illustration below describes the interception of solar radiation by the Earth. In this system, the Earth and Sun, the parts or component blocks, are represented by two colored circles of different size. The process of solar emission and the interception of the Sun's emitted radiation by the Earth (the connection) is illustrated by the drawn lines.

Figure 9.9: Simple Visual Model of Solar Radiation being Emitted from the Sun and Intercepted by the Earth



Characteristics of a System

Most systems share the same common characteristics. These common characteristics include the following:

1. Systems have a structure that is defined by its parts and processes.
2. Systems are generalizations of reality.
3. Systems tend to function in the same way. This involves the inputs and outputs of material (energy and/or matter) that is then processed causing it to change in some way.
4. The various parts of a system have functional as well as structural relationships between each other.
5. The fact that functional relationships exist between the parts suggests the flow and transfer of some type of energy and/or matter.
6. Systems often exchange energy and/or matter beyond their defined boundary with the outside environment, and other systems, through various input and output processes.
7. Functional relationships can only occur because of the presence of a driving force.
8. The parts that make up a system show some degree of integration - in other words the parts work well together.

Properties of a System

Within the boundary of a system we can find three kinds of properties:

1. Elements are the kinds of parts (things or substances) that make up a system. These parts may be atoms or molecules, or larger bodies of matter like sand grains, rain drops, plants, animals, etc.

Notes

2. Attributes are characteristics of the elements that may be perceived and measured. For example: quantity, size, color, volume, temperature, and mass.
3. Relationships are the associations that occur between elements and attributes. These associations are based on cause and effect.

> *Notes* We can define the state of the system by determining the value of its properties (the elements, attributes, and/or relationships).

Types of Systems

Scientists have examined and classified many types of systems. Some of the classified types include:

1. ***Isolated System:*** It is a system that has no interactions beyond its boundary layer. Many controlled laboratory experiments are this type of system.
2. ***Closed System:*** It is a system that transfers energy, but not matter, across its boundary to the surrounding environment. Our planet is often viewed as a closed system.
3. ***Open System:*** It is a system that transfers both matter and energy can cross its boundary to the surrounding environment. Most ecosystems are example of open systems.
4. ***Morphological System:*** This is a system where we understand the relationships between elements and their attributes in a vague sense based only on measured features or correlations. In other words, we understand the form or morphology a system has based on the connections between its elements. We do not understand exactly how the processes work to transfer energy and/or matter through the connections between the elements.
5. ***Cascading System:*** This is a system where we are primarily interested in the flow of energy and/or matter from one element to another and understand the processes that cause this movement. In a cascading system, we do not fully understand quantitative relationships that exist between elements related to the transfer of energy and/or matter.
6. ***Process-response System:*** This is a system that integrates the characteristics of both morphological and cascading systems. In a process-response system, we can model the processes involved in the movement, storage, and transformation of energy and/or matter between system elements and we fully understand how the form of the system in terms of measured features and correlations.
7. ***Control System:*** A system that can be intelligently manipulated by the action of humans.
8. ***Ecosystem:*** It is a system that models relationships and interactions between the various biotic and abiotic components making up a community or organisms and their surrounding physical environment.

There are four basic types of system depending on whether the parts and the whole can display choice, and therefore, be purposeful. The four kinds of system are shown in Table 9.1

Notes

Figure 9.10: Types of Systems

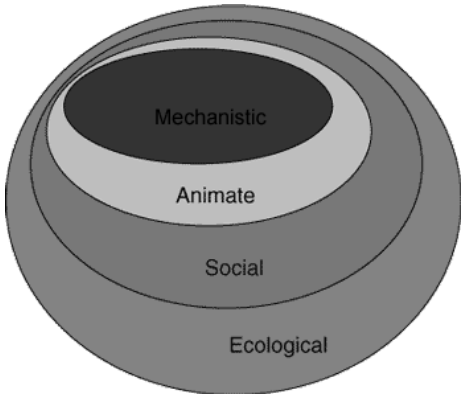


Table 9.1: Four Types of Systems

Types of System Model	Parts	Whole	Example
Mechanistic	No choice	No choice	Machines
Animate	No choice	Choice	Persons
Social	Choice	Choice	Corporations
Ecological	Choice	No Choice	Nature

These type form a hierarchy with ecological systems – the highest type. All but mechanistic systems can incorporate as parts other systems of the same or a lower type, but not of a higher type; for example, social systems (e.g., society) may incorporate animate systems (people) and mechanistic systems (machines), but a mechanistic system cannot incorporate either an animate or social system. Ecological systems can incorporate systems of all the other types.

Only animate and social systems can be said to be purposeful.

Now consider each type of system in a bit more detail.

1. ***Mechanistic Systems:*** Mechanistic systems and their parts have no purposes of their own, but their essential parts make possible the functioning of the whole. All mechanisms are mechanistic systems. Plants are also. Clocks are common examples of such systems; they operate with a regularity dictated by their internal structure and the causal laws of nature. Neither the whole nor the parts of a clock display choice, but they have functions. Similarly, an automobile is a mechanical system that has no purpose of its own but serves its driver's and passengers' purposes. In addition, an automobile's fuel pump (a mechanical system) has the function of supplying its fuel injector or carburetor with fuel, without which the automobile could not carry out its defining function.

   Mechanistic systems are either open or closed, closed if their behavior is unaffected by any external conditions or events; open if they are so affected. The universe was conceptualized by Newton as a closed (self-contained) mechanical system, with no environment-like a hermetically sealed clock. On the other hand, the planet Earth is seen as an open system, one whose motion is influenced by other bodies in the solar system.

2. ***Animate Systems:*** These are conceptualized as purposeful systems whose parts have no purposes of their own. The principal purpose of such systems is survival. A person's lungs have no purpose of their own; but they function to enable a person to extract oxygen from the environment so as to survive. Animate systems are necessarily open; they must interact

Notes

with their environments in order to survive Understanding these interactions are essential for understanding their properties and behavior.

Animate systems are living systems. "Life" has been defined in many different Ways. The definition now most widely accepted by biologists involves the ways concept autopsies:

"The maintenance of units and wholeness, while components themselves are being continuously or periodically disassembled and rebuilt, created and decimated, produced and consumed." (Zeleny, 1981, p. 5)

From this definition it follows that social and ecological systems are also alive. (Many biologists are unhappy about this consequence of their definition of 'life.')

3. ***Social Systems:*** These are systems that (1) have purposes of their own, (2) consist of parts at least some of which are animate, hence have purposes of their own, and (3) are a part of one or more larger (containing) systems that may have purposes of their own and that may contain other social systems. For example, a local government viewed as a social system is part of a state government, which is also a social system, This, in turn, is part of a national government. Social systems can be and usually are nested.

4. ***Ecological Systems:*** Such systems contain mechanistic, animate, and social systems as parts and, therefore, containing some parts that have purposes of their own. However, these systems as a whole are conceptualized as having no purpose of their own. Nature, of course, is commonly taken to be an ecological system as is our environment.

Ecological systems serve the purposes of their animate and social parts, and provide necessary inputs to these and open deterministic systems. They also provide a receptacle for their waste as well as their useful products. Such service and support is their function. An ecological system can be affected mechanistically by the mechanical or purposeful behavior of its parts. For example, the purposeful use by people of fluorocarbons as a propellant and the emissions of power plants affect the ozone layer mechanistically.

Animate and social systems are frequently confronted with situations in which their choices can affect their effectiveness, either positively or negatively. Such situations are problematic.

Task Problems are situations in which a system's choice can make a significant difference to that system. Analyze.

Modeling and Simulation Process

An outline of the essential steps involved in carrying out a modelling and simulation study is provided in the discussion that follows. Although the initial steps can be effectively presented using various notions that have been previously introduced, there are several aspects of the latter stages that require extensive elaboration. This is provided in the discussions that follow. An overview of the process is provided in Figure 9.11.

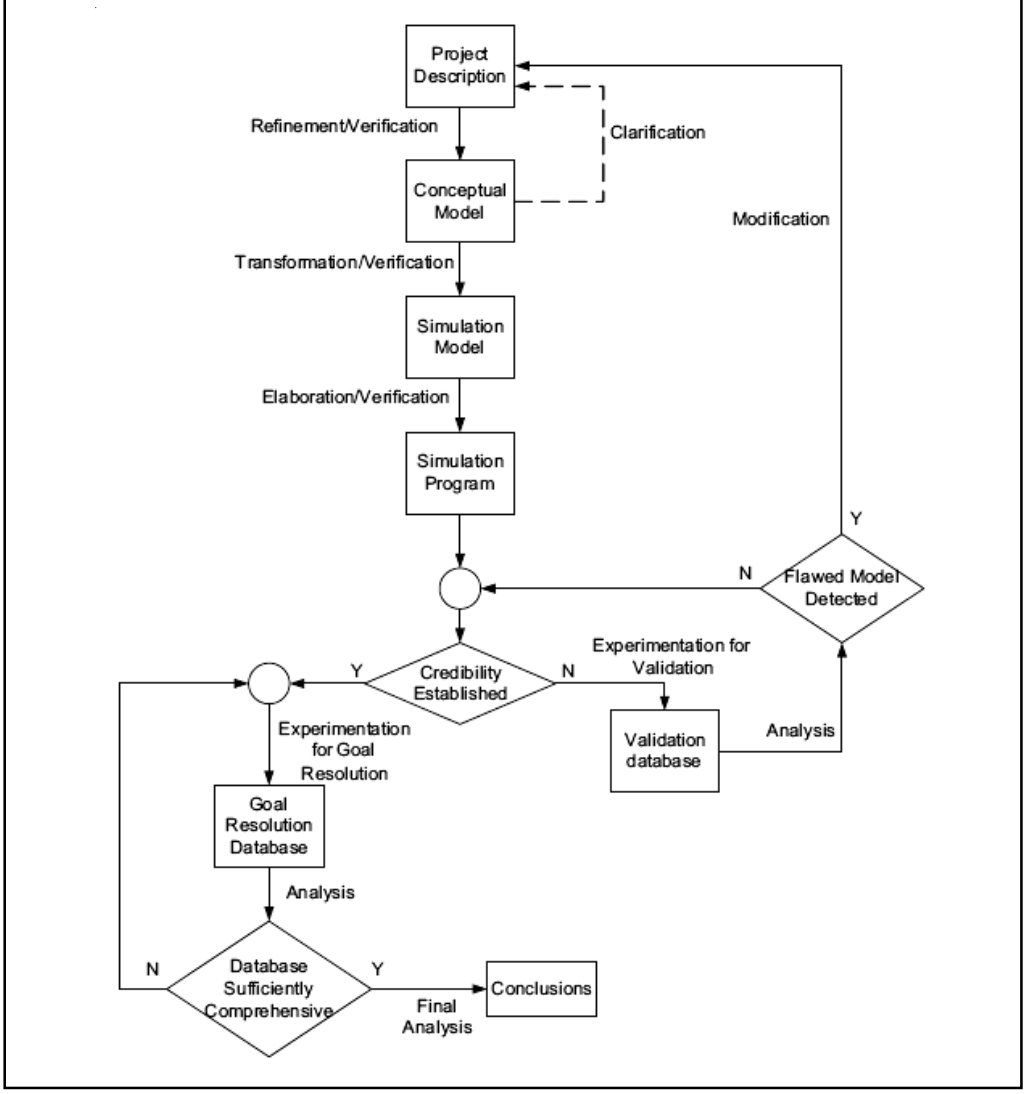
The overview of Figure 9.11 does not include a preliminary phase during which solution alternatives for the problem are explored and a decision is made to adopt a modelling and simulation approach. We note that the option of also carrying out other complementary approaches is entirely reasonable and is often prudent for some portions of the problem. Although this preliminary phase is not explicitly represented in Figure 9.11 its existence and importance must nevertheless be recognised.

It should be emphasised that a modelling and simulation project of even modest size is often carried out by a team of professionals where each member of the team typically contributes

Notes

some special expertise. There is, therefore, a need for communication among team members. Some facets of the discussion have their basis in this communication requirement.

Figure 9.11: Modeling and Simulation Process



Project Description

The process begins with the preparation of a document called the project description. This document includes a statement of the project goal(s) and a description of those behavioural features of the SUI that have relevance to the goals. These behaviour features are typically formulated in terms of the various entities that populate the space that the SUI embraces with particular focus on the interactions among these entities. It is, more or less, an informal description inasmuch as it relies mainly on the descriptive power of natural language. The language is, furthermore, often heavily coloured with jargon associated with the SUI. This jargon may not be fully transparent to all members of the project team and this can contribute to both an inherent lack of precision and, as well, to communication problems.

With few exceptions, the SUI also has structural features that provide the context for the interactions among the entities (e.g., the layout of the pumps at a gas station or the topology of the network

Notes

of streets being serviced by a taxi company). Informal sketches are often the best means of representing these structural features. These are an important part of the presentation because they provide a contextual elaboration that can both facilitate a more precise statement of the project goals and as well, help to clarify the nature of the interaction among the entities. Because of these contributions to understanding, such sketches are often a valuable component of the project description.

Conceptual Model

The information provided by the project description is, for the most part, unstructured and relatively informal. Because of this informality it is generally inadequate to support the high degree of precision that is required in achieving the objective of a credible model embedded within a computer program. A refinement phase must be carried out in order to add detail where necessary, incorporate formalisms wherever helpful, and generally enhance the precision and completeness of the accumulated information. Enhanced precision is achieved by moving to a higher level of abstraction than that provided by the project description. The reformulation of the information within the project description in terms of parameters and variables is an initial step because these notions provide a fundamental means for removing ambiguity and enhancing precision. They provide the basis for the development of the simulation model that is required for the experimentation phase.

There is a variety of formalisms that can be effectively used in the refinement process. Included here are mathematical equations and relationships (e.g., algebraic and/or differential equations), symbolic/graphical formalisms (e.g., Petri nets, finite state machines), rule based formalisms, structured pseudo code, and combinations of these. The choice depends on suitability for providing clarification and/or precision. The result of this refinement process is called the conceptual model for the modelling and simulation project. The conceptual model may, in reality, be a collection of partial models each capturing some specific aspect of the SUI's behaviour. The representations used in these various partial models need not be uniform.

The conceptual model is a consolidation of all relevant structural and behavioural features of the SUI in a format that is as concise and precise as possible. It provides the common focal point for discussion among the various participants in the modelling and simulation project. In addition, it serves as a bridge between the project description and the simulation model that is essential for the experimentation activity (i.e., the simulation phase). As we point out below, the simulation model is a software product and its development relies on considerable precision in the statement of requirements. One of the important purposes of the conceptual model is to provide the prerequisite guidance for the software development task.

In Figure 9.11, a verification activity is associated with the transition from the project description to the conceptual model. As will transition under consideration because it involves a reformulation of the key elements of the model from one form to another and the integrity of this transformation needs to be confirmed.

In the modelling and simulation literature, the phrase 'conceptual model' is frequently reduced simply to 'model'. Our usage of the word 'model' without a modifier generally implies a composite notion that program successors where the latter two notions are described in the discussion that follows.

As a concluding observation in this discussion, it is worth pointing out that there is by no means a common understanding in the modelling and simulation literature of the nature and role of a conceptual model. The overview presented by Robinson gives considerable insight into the various perspectives that prevail.

Notes

Simulation Model

The essential requirement for the experimentation phase of a modeling and simulation project is an executable computer program that embodies the conceptual model. It evolves from a transformation of the conceptual model into a representation that is consistent with the syntax and semantic constraints of some programming language. This program is the simulation model for the project. It is the execution of this program (or more correctly), Enhanced version of it; see following and simulation project is an executable computer program that embodies the system under investigation. The solution to the underlying problem is the simulation model for the project. It is the execution of this program that is embedded in the project goal(s) is obtained from the data reflected in this behaviour. As will the conceptual model. It evolves from a transformation of a conceptual model and its simulation model and simulation appeared in recent years; some examples are: SIMSCRIPT II.5, MODSIM, GPSS, SIMAN, ACSL, Modelica, Arena, CSIM, and SIMPLE ++. Such languages generally provide features to support the management of time, collection of data, and presentation of required output information. In the case of projects in the DEDS domain, additional features for the generation of random variates, management of queues, and the statistical analysis of data are also provided.

The simulation model is the penultimate stage of a development process that began with the decision to formulate a modelling and simulation project to resolve an identified problem. The Simulation model is a software product and as such, the process for its development shares many of the general features that characterise the development of any software product.

Note that in Figure 9.11 the transition from the conceptual model to the simulation model is associated with two activities: namely, transformation and verification.

Notes As in the earlier transition from project description to conceptual model, verification is required here to confirm that the transformation has been correctly carried out.

Simulation Program

The outline of the simulation model provided above is idealised inasmuch as it suggests that the simulation model is directly capable of providing the behaviour-generating mechanism for the simulation activity. In reality this program code segment is never self-sufficient and a variety of auxiliary services must be superimposed. The result of augmenting the simulation model with complementary program infrastructure that provides these essential functional services is the simulation program.

The services in question fall into two categories: one relates to fundamental implementation issues whereas the other is very much dependent on the nature of the experiments that are associated with the realisation of the project goals. Included within the first category are such basic tasks as initialisation, control of the observation interval, management of stochastic features (when present), solution of equations (e.g., the differential equations of a continuous system model), data collection, and so on. Convenient programming constructs to deal with these various tasks are normally provided in software environments specifically designed to support the simulation activity. But this is certainly not the case in general-purpose programming environments where considerable additional effort is often required to provide these functional requirements.

The second category of functional services can include such features as data presentation (e.g., visualisation and animation), data analysis, database support, optimisation procedures, and the like. The extent to which any particular modelling and simulation project requires services from

Notes

this second category can vary widely. Furthermore, modelling and simulation software environments provide these services only to varying degrees and consequently, when they are needed; care must be taken in choosing an environment that is able to deliver the required services at an adequate level.

The manner in which the support services to augment the simulation model are invoked varies significantly among software environments. Almost always there is at least some set of parameters that need to be assigned values in order to choose from available options. Often some explicit programming steps are needed. Considerable care must be taken when developing the simulation program to maintain a clear demarkation between the code of the simulation model and the code required to invoke the ancillary services. Blurring this separation can be detrimental because the resulting simulation program may become difficult to verify, understand, and/or maintain. It has, in fact, been frequently noted that an important quality attribute of a simulation software platform is the extent to which it facilitates a clear separation of the code for the simulation model from the infrastructure code required for the experimentation that is required for the achievement of the project goal(s).

Figure 9.11 indicates that a verification activity needs to be carried out in the transition from the simulation model to the simulation program. This need arises because this transition typically involves a variety of decisions relating to the execution of the simulation model and the correctness of these decisions must be confirmed. Consider, for example, a simulation model that incorporates a set of ordinary differential equations. Most modelling and simulation programming environments offer a variety of solute on methods for such equations and each has particular strengths and possibly weaknesses as well. If the equations in question have distinctive properties, then there exists a possibility of an improper choice of solution method. The verification process applied at this stage would uncover the existence of such a flaw when it exists.

Operational Phases

Thus far our outline of the modelling and simulation process has focused on the evolution of a series of interdependent representations of SUI. However, with the existence of the simulation program, the stage is set for two operational phases of the process that we now examine. The first of these is the validation phase whose purpose is to establish the credibility of each of the model realisations, from the perspective of the project goals.

The second phase, which can begin only after the model's credibility has been established, is the experimentation phase, or more specifically, the simulation phase. This activity is presented in Figure 9.11 as the task of 'goal resolution'. This is achieved via a sequence of experiments with the simulation program during which an ever-increasing body of data is collected and analysed until it is apparent that a 'goal resolution database' is sufficiently complete and comprehensive to permit conclusions relating to the goal(s) to be confidently formulated.

Concept of the Environment

Intuitively, the notion of "the environment'' in AI and robotics refers to the relatively enduring and stable set of circumstances that surround some given individual. My environment is probably not the same as yours, though they may be similar. On the other hand, although my environment starts where I leave off (at my skin, perhaps), it has no clear ending-point. Nor is it necessarily defined in terms of metric space; if physically distant circumstances have consequences for my life (via the telephone, say) then they are properly regarded as part of my environment as well. The environment is where agents live, and it determines the effects of their actions. The environment is thus a matter of importance in computational modeling; only if we know what an agent's environment is like can we determine if a given pattern of behavior is adaptive. In

Notes

particular we need a positive theory of the environment, that is, some kind of principled characterization of those structures or dynamics or other attributes of the environment in virtue of which adaptive behavior is adaptive.

Herbert Simon discussed the issue in his pre-AI work. His book Administrative Behavior, for example, presents the influential theory that later became known as limited rationality. In contrast to the assumption of rational choice in classical economics, Simon describes a range of cognitive limitations that make fully rational decision-making in organizations impracticable. Yet organizations thrive anyway, he argues, because they provide each individual with a structured environment that ensures that their decisions are good enough. The division of labor, for example, compensates for the individual's limited ability to master a range of tasks. Structured flows of information, likewise, compensate for the individual's limited ability to seek this information out and judge its relevance. Hierarchy compensates for the individual's limited capacity to choose goals. And fixed procedures compensate for individuals' limited capacity to construct procedures for themselves.

In comparison to Simon's early theory in Administrative Behavior, AI has downplayed the distinction between agent and environment. In Newell and Simon's early work on problem solving, the environment is reduced to the discrete series of choices that it presents in the course of solving a given problem. The phrase "task environment" came to refer to the formal structure of the search space of choices and outcomes. This is clearly a good way of modeling tasks such as logical theorem-proving and chess, in which the objects being manipulated are purely formal. For tasks that involve activities in the physical world, however, the picture is more complex. In such cases, the problem solving model analyzes the world in a distinctive way. Their theory does not treat the world and the agent as separate constructs. Instead, the world shows up, so to speak, phenomenological: in terms of the differences that make a difference for this agent, given its particular representations, actions, and goals. Agents with different perceptual capabilities and action repertoires, for example, will inhabit different task environments, even though their physical surroundings and goals might be identical.

Newell and Simon's theory of the task environment, then, tends to blur the difference between agent and environment. As a framework for analysis, we find the phenomenological approach valuable, and we wish to adapt it to our own purposes. Unfortunately, Newell and Simon carry this blurring into their theory of cognitive architecture. They are often unclear whether problem solving is an activity that takes place wholly within the mind, or whether it unfolds through the agent's potentially complicated interactions with the physical world. This distinction does not arise in cases such as theorem-proving and chess, or in any other domain whose workings are easily simulated through mental reasoning. But it is crucial in any domain whose actions have uncertain outcomes. Even though we wish to retain Newell and Simon's phenomenological approach to task analysis, therefore, we do not wish to presuppose that our agents reason by conducting searches in problem spaces. Instead, we wish to develop an analytical framework that can guide the design of a wide range of agent architectures. In particular, we want an analytical framework that will help us design the simplest possible architecture for any given task.

Continues and Discrete Systems

In models for discrete event dynamic systems (i.e., DEDS models) state changes occur at particular points in time whose values are not known a priori. As a direct consequence, (simulated) time advances in discrete 'jumps' that have unequal length.

In contrast, with models that emerge from the domain of continuous time dynamic systems (i.e., CTDS models), state changes occur continuously (at least in principle) as time advances in a

Notes

continuous fashion over the length of the observation interval. It must, however, be realities introduced by the computational process. It is simply infeasible for any practical procedure to actually yield data at every value of time within the continuum of the observation interval. Thus, from the perspective of the observer, state changes do apparently occur with discrete 'jumps' as the solution unfolds over the observation interval. Our presentation in this textbook may give the erroneous impression that models neatly separate into the two broad categories that we refer to as DEDS models and CTDS models. This is an oversimplification. There is, in fact a third category of models that are usually called combined models where the name reflects the combination of elements from both the discrete and continuous domains. As an illustration consider the parts in a manufacturing plant that move from one workstation to another on the way to assembly into a final product. At these workstations, queues form and the service function provided by the workstation may have random aspects (or may become inoperative at random points in time). Thus the basic elements of a DEDS model are present. At some workstations the operation may involve heating the part to a high temperature in a furnace. This heating operation and the control of it would best fall in the realm of a CTDS model. Hence the overall model that is needed has components from the two basic domains.

Work on the development of modelling formalisms and tools for handling this third category of combined models has a long history. The interested reader wishing to explore this topic in more detail will find relevant discussions in Cellier, Ören and Praehofer, the initial portion of a simulation experiment.

Modern systems that may be found in various domains like automotive, defense, medical and communications, integrate continuous and discrete models. In a recent ITRS study covering the domain of mixed continuous discrete systems, the conclusion is a "shortage of design skills and productivity arising from lack of training and poor automation with needs for basic design tools" as one of the most daunting challenges in this domain (ITRS, 2003). One of the main difficulties in the definition of CAD tools for Continuous/Discrete (C/D) systems is due to the heterogeneity of concepts manipulated by the discrete and the continuous components. Therefore, in the case of validation tools, several execution semantics have to be taken in consideration in order to perform global simulation:

In Discrete Models (DM), the time represents a global notion for the overall system and advances discretely when passing by time stamps of events, while in Continuous Models (CM), the time is a global variable involved in data computation and it advances by integration steps that may be variable.

In discrete models, processes are sensitive to events while in continuous models processes are executed at each integration step.

Currently, co-simulation is a popular validation technique for heterogeneous systems. This technique was successfully applied for discrete systems, but very few applied it for C/D systems. The co-simulation allows joint simulation of heterogeneous components. This requires the elaboration of a global execution model, where the different components communicate through a co-simulation bus via simulation interfaces performing adaptation. For C/D systems co-simulation, the simulation interfaces have to provide efficient synchronization models in order to cope with the heterogeneous.

This Unit presents CODIS (Continuous/Discrete Systems simulation), a co-simulation framework for C/D systems validation. This framework assists designers in building global simulation models. The supported simulators are Simulink for continuous models and OSCI SystemC simulator for discrete models.

9.5 CODIS Framework

Notes

Synchronization and Generic Architecture for C/D Simulation in CODIS

For an accurate synchronisation, each simulator involved in a C/D co-simulation must consider the events coming from the external world and it must reach accurately the time stamps of these events. We refer to this as events detection. These time stamps are the synchronization and communication points between the simulators involved in the co-simulation. Therefore, the continuous simulator, Simulink, must detect the next discrete event (timed event) scheduled by the discrete simulator, once the latter has completed the processing corresponding to the current time. In case of SystemC, these events are: clock events, timed notified events, events due to the wait function. This detection requires the adjustment of integration steps in Simulink (see Figure 9.14). The discrete simulator, SystemC, must detect the state events. A state event is an unpredictable event, generated by the continuous simulator, whose time stamp depends on the values of the state variables (ex: a zero-crossing event, a threshold overtaking event, etc.). This implies the control of the discrete simulator advancement in time: in stead of advancing with a normal simulation step, the simulator has to advance precisely until the time stamp of the state event (see Figure 9.12).

Figure 9.12: C/D Synchronisation in CODIS

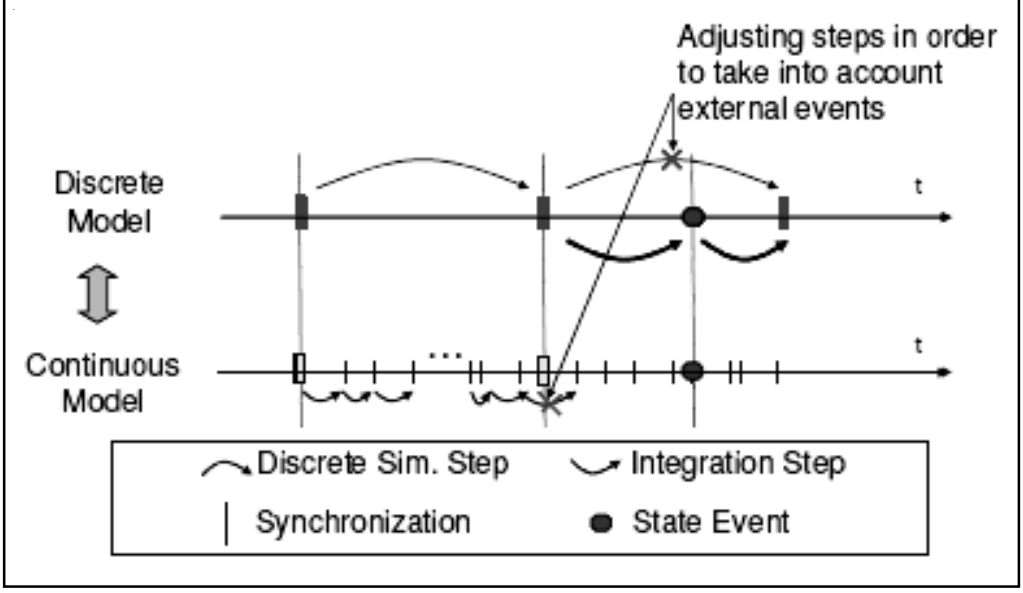


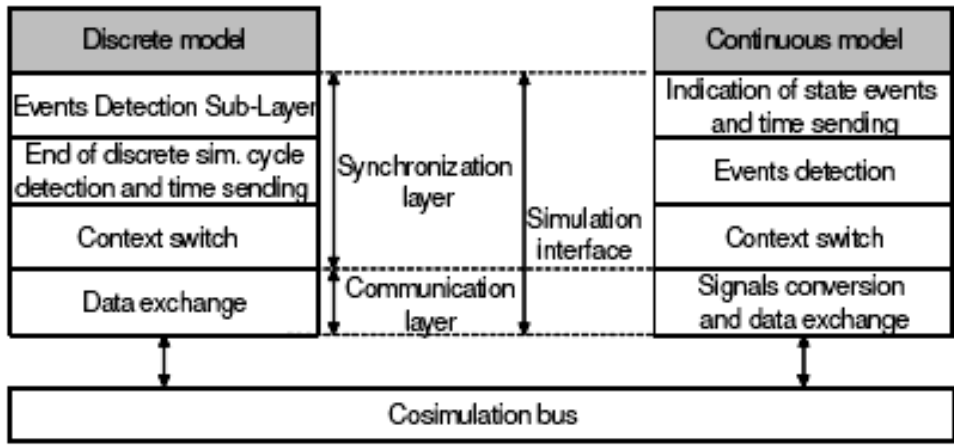
Figure 9.13 illustrates the generic architecture used in CODIS for the C/D simulation. CM and DM communicate through a co-simulation bus via simulation interfaces. Each simulation interface presents two main layers:

The synchronization layer provides the synchronisation requirements discussed above for both CM and DM; this layer is composed of three sub-layers, each of them achieving an elementary functionality for synchronisation.

The communication layer is in charge of sending/receiving data between CM and DM. More details on synchronization and CODIS simulation architecture may be found in Bouchimma *et al.*, 2005.

Notes

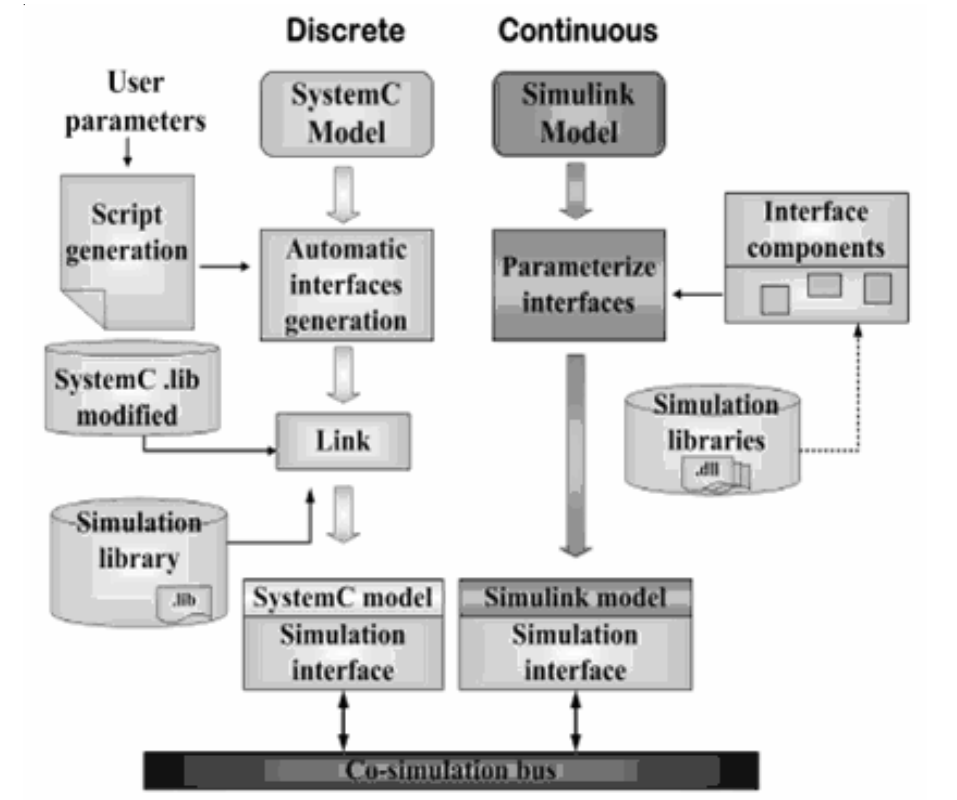
Figure 9.13: Generic Architecture for Accurate C/D Simulation



Simulation Model Generation in CODIS

Based on the presented synchronization model and the generic architecture, a flow for automatic generation of simulation models was implemented in CODIS (see Figure 9.14). The inputs of the flow are the CM in Simulink and the DM in System C. The output of the flow is the global simulation model, including the co-simulation bus and the simulation interfaces.

Figure 9.14: Flow for Simulation Models Generation



Notes

The interfaces are generated by composing elements from the CODIS library. These elements implement the layers in Figure. They may be customized in terms of number of ports and their data type. The interfaces for DM are automatically generated by a script generator that has as input user defined parameters. The model is compiled and the link editor calls the library from SystemC and a static library called "simulation library" (see Figure 9.14). The interfaces for Simulink are functional blocks programmed in C++ using S-Functions. These blocks are manipulated like all other components of the Simulink library. The user starts by dragging the interfaces from the library into the model's window, then parameterizes them and finally connects them to the model inputs and outputs. Before the simulation, the functionalities of these blocks are loaded by Simulink from the ".dll" dynamically linked libraries.

Experimental Results

To evaluate the performances of simulation models generated in CODIS, we measured the overhead given by the simulation interfaces. The overhead caused by the Simulink integration step adjustment when detecting a SystemC event has been measured in a maximum of 10% of total simulation time. The overhead caused by IPC (Inter Process Communication) used for the context switch and the communication layers has been measured in order of maximum 20% of the total simulation time. The cost of the added synchronization functionality in the case of SystemC is negligible and does not exceed 0.02% of the total simulation time.

Tasks

1. What was the name of the primarily tool which use as the solution of differential equations system?
2. What is the role of modelling and simulation?

Self Assessment

Fill in the blanks:

11. The correct formulation of a …… can be a challenging task and its correctness is essential to the quality of the results flowing from the modeling and simulation project.
12. The ……. layer is in charge of sending/receiving data between CM and DM.
13. In discrete models, processes are sensitive to events while in continuous models processes are executed at each …….. step.
14. In the modelling and simulation literature, the phrase '……..' is frequently reduced simply to 'model'.
15. A data requirement here could be a single …….. value representing the coefficient of friction associated with the flow.

Notes

Caselet

ICAR Plans to Adopt GPS Tech for Appropriate Farming Techniques

The Indian Council of Agricultural Research is working on a scheme to increase agricultural productivity by providing exact information to farmers on the type of crops they should grow, fertilizer and water requirements through the use of GPS technology.

Under the project, ICAR has collected global positioning system (GPS)-based soil samples across the country on the nutrient status and acidic reaction of soil in agri-regions to build a database on the appropriate farming methods to be employed for maximum productivity.

Subsequently, information on appropriate farming techniques to be adopted in specific areas, as identified through the GPS-based soil samples, will be disseminated to farmers over the internet.

The Project Directorate for Farming System Research (PDFSR), Meerut, an institute of ICAR, is already implementing the initiative on a pilot basis in two villages of Western Uttar Pradesh — Jainpur in Meerut and Matiala in Ghaziabad.

"The agricultural land will be divided into homologous zones on the basis of the requirement of fertilizers, the data along with the map of the particular agricultural field will be available on the internet which could be easily accessed by the farmers," the Project Director of PDFSR, Modipuram, Meerut, Mr B. Gangwar, said.

According to scientists, the requirement of fertilizers in the soil depends on their qualities, such as the extent of acidity & alkalinity, PH value, humus content and the type of crops harvested and sown which largely varies.

Over-use of fertilisers is turning the soil infertile, which is a major concern, Mr Gangwar said.

"The data will be updated after every crop," he added.

In addition to GPS, Global Information System (GIS) and Simulation Modelling for Decision Support System (DSS) will be utilised for this project. The farmers will also be able to know the type of crop suitable for their fields.

The results of the pilot projects in Western UP are encouraging, Mr Gangwar said.

"In India, agricultural land is shrinking rapidly so we have no other option but to conserve and utilise the natural resources for maximum productivity by using appropriate techniques of farming system," he said.

According to government estimates, small and marginal farmers will hold more than 91 per cent of farm holdings by 2030.

ICAR scientists have also developed an integrated farming system, which increases the productivity of water by up to four times and in this system, rice cultivation is integrated with pisciculture and poultry farming.

Notes

9.6 Summary

- Systems Engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation.
- Systems Engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs.
- System models are hierarchical or layered as a system is represented at different levels of abstraction. The top level of hierarchy presents the complete system.
- The data objects, functions, behaviors are represented. As the hierarchy is refined or layered, component level detail is modeled and finally system models evolve into engineering models.
- Good system engineering begins with a clear understanding of context – the world view – and then progressively narrows focus until technical details are understood.
- System engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy.
- This unit introduced the main concepts of SYSTEM SIMULATION to model dynamic systems by identifying variables and causal affects.
- The model was parameterized with data and equations and simulated over time.
- Simulation results can be viewed in a Data File editor, exported to other applications or presented with configurable tables, charts and graphs.

9.7 Keywords

CM: Continuous Models

DEDS: Discrete event dynamic system

DM: Discrete Models

SUI: System Under Investigation

Vector Variable: A collection of variables organized as a linear array is called a vector variable.

9.8 Review Questions

1. Explain the meaning of the term:

   (a) Isolated System (b) Closed System

   (c) Morphological System (d) Cascading System

2. What are common characteristics of a system?

3. A new apartment building is being designed. It will have ten floors and will have six apartments on each floor. There will, in addition be two underground parking levels. The developer needs to make a decision about the elevator system that is to be installed in the building. The choice has been reduced to two alternatives: either two elevators each with a capacity of 15 or three smaller elevators

   (a) Develop a list of possible performance criteria that could be used to evaluate the relative merits of the two alternative designs.

Notes

(b) Develop a list of behavioral rules that would be incorporated in the formulation of the model of the elevator system (e.g., when does an elevator change its direction of motion, which of the elevators responds to a particular call for service, where does an idle elevator 'park,' etc.?)

(c) A model's behaviour can 'unfold' only as a consequence of input data. Develop a list of input data requirements that would necessarily become an integral part of a study of the elevator system, for example, arrival rates of tenants at each of the floors, velocity of the elevators, and so on.

4. What are the types of systems?
5. What is modeling? Explain in detail.
6. Describe, in brief, the events and states in terms of simulation.
7. What is the difference between object model and inheritance model?
8. Modeling and Simulation is a discipline, it is also very much an art form. Explain.
9. Simulators represent an application area of the modelling and simulation paradigm. Analyze.
10. A model plays the role of a surrogate for the system it represents. Discuss.

Answers: Self Assessment

1.	Systems	2.	construction
3.	parts	4.	common
5.	Architectural	6.	Data flow diagrams
7.	UML	8.	Model
9.	Modelling	10.	Simulator
11.	data model	12.	communication
13.	integration	14.	conceptual model
15.	scalar		

9.10 Further Readings

Books

Balci, O.,(1994), Validation, verification, and testing techniques throughout the life cycle of a simulation study, Annals of Operations Research, 53: 121–173.

Balci, O., (2001), A methodology for certification of modeling and simulation applications, ACM Transactions on Modeling and Computer Simulation, 11: 352–377.

Birta, L.G. and Ozmizrak, N.F., (1996), A knowledge-based approach for the validation of simulation models: The foundation, ACM Transactions on Modeling and Computer Simulation, 6: 67–98.

Boehm, B.W., (1979), Software engineering: R&D trends and defense needs, in: P. Wegner (Ed.), Research Directions in Software Technology, MIT Press, and Cambridge, MA.

Notes

Cellier, F.E., (1986), Combined discrete/continuous system simulation – application, techniques and tools, in Proceedings of the 1986 Winter Simulation Conference.

Department of Defense (DoD) Recommended Practices Guide (RPG) for Modeling and Simulation VV&A, Millennium Edition (available at http://vva.dmso.mil).

General Accounting Office, (1976), Report to the Congress: Ways to improve management of federally funded computerized models, report LCD-75-111, U.S. General Accounting Office, Washington, DC.

General Accounting Office, (1979), Guidelines for model evaluation, report PAD-79-17, U.S. General Accounting Office, Washington, DC.

General Accounting Office, (1987), DOD simulations: Improved assessment procedures would increase the credibility of results, report GAO/PEMD-88-3, U.S. General Accounting Office, and Washington, DC.

Ören, T.I., (1981), Concepts and criteria to access acceptability of simulation studies, Communications of the ACM, 24: 180–189.

Online links

http://en.wikipedia.org/wiki/Systems_engineering

http://www.sie.arizona.edu/sysengr/whatis/whatis.html

Notes

Unit 10: Creating an Architectural Design

CONTENTS

Objectives

Introduction

10.1 Data Design

10.2 Data Design at Component Level

10.3 Architectural Design

10.3.1 Representing the System in Context

10.3.2 Defining Archetypes

10.3.3 Refining the Architecture into Components

10.3.4 Describing Instantiations of the System

10.4 Summary

10.5 Keywords

10.6 Review Questions

10.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Comprehend the Data Design
- Recognize the architectural design

Introduction

As the size of software systems enlarges, the algorithms and data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organization of the overall system – the software architecture – presents a new set of design problems. This level of design has been addressed in a number of ways including informal diagrams and descriptive terms, module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component integration mechanisms.

Software architecture has started to receive focus since the last decade. It is becoming clearer that an early and careful architecture design can greatly reduce the failure rates of software projects. A good architecture design partitions the functional requirements of a software system into a manageable set of interacting components. Quality attributes such as efficiency, usability, modifiability, reliability, and security can be verified and estimated over the architecture design before any code is produced. As the blueprint of a software system, an architecture design allows system analysts and software architects to communicate effectively with stakeholders. It also sets the grounds for the subsequent software design process. The software design process furnishes the details of each component. Then the detailed design can be implemented via coding, which is followed by debugging, testing, and maintenance.

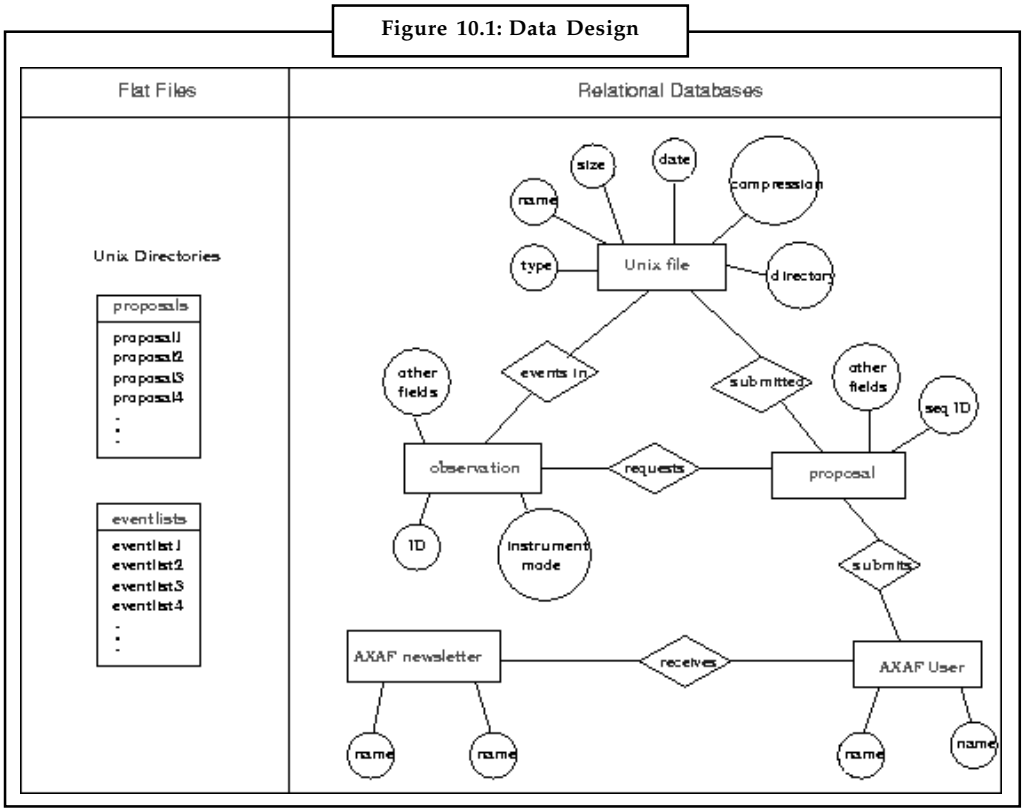
10.1 Data Design

Notes

Data such as event lists and images need to be kept in files as they are received. They also need to be correlated with engineering and other ancillary data which arrive as a continuous time stream and to be associated with a calibration test or an observation ID. A level of isolation between the data and users is desirable for security, performance and ease of administration. The following design was chosen. Files are kept in simple directory structures. Metadata about the files, extracted from file headers or supplied by the archiving process, is stored in databases. This allows file searches on the basis of their contents. Continuous time data extracted from engineering files is also stored in databases so the correct values can be easily associated with an image or an event list with defined time boundaries. In addition to partial or entire file contents, file external characteristics such as its location in the archive, size, compression, creation time are also stored in databases for archive internal use.

Notes In addition to databases with contents originating in files, there are also databases which are updated independently, such as the AXAF observing catalog and the AXAF users database.

Figure 10.1: Data Design



A naive instance of the data design is shown on Figure 10.1. The archive contains a number of proposal files submitted by users. It also contains a number of event files, products of observed data processing. A table in a database contains the characteristics of each file. The proposal table contains a record for each proposal which points to the associated file in the file table. The observation table contains a record for each observed target and has a pointer to the associated proposal. An observation is also associated with files in the file table which contain observed

Notes

events. Related to the proposal is the AXAF user who submitted it and for whom there is an entry in the AXAF user table. An AXAF user may have a subscription to the AXAF newsletter.

An activity that translates the data model developed during analysis into implementable data structures.

Data design translates data objects defined during analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

There are small and large businesses that contain lot of data. There are dozens of databases that serve many applications comprising of lots of data. The aim is to extract useful information from data environment especially when the information desired is cross functional.

Techniques like data mining are used to extract useful information from raw data. However, data mining becomes difficult because of some factors:

- Existence of multiple databases.
- Different structures.
- Degree of detail contained with databases. Alternative solution is concept of data warehousing which adds an additional layer to data architecture. Data warehouse encompasses all data used by a business.

Did u know? **What is data warehouse?**

A data warehouse is a large, independent database that serves the set of applications required by a business. Data warehouse is a separate data environment.

Self Assessment

Fill in the blanks:

1. A level of ……. between the data and users is desirable for security, performance and ease of administration.
2. A good architecture design partitions the …….. requirements of a software system into a manageable set of interacting components.
3. …….. about the files, extracted from file headers or supplied by the archiving process, is stored in databases.
4. Techniques like …….. are used to extract useful information from raw data.
5. Data mining tools predict future trends and behaviors, allowing businesses to make proactive, …….. decisions.
6. A data warehouse is a …….. database that is designed for query and analysis rather than for transaction processing.
7. Data warehouses are designed to help you …….. data.
8. Data warehouses must put data from disparate sources into a ……. format.

10.2 Data Design at Component Level

It focuses on representation of data structures that are directly accessed by one or more software components. Set of principles applicable to data design are:

Notes

- Systematic analysis principles applied to function and behavior should also be applied to data.
- All data structures and operations to be performed on each should be identified.
- The content of each data object should be defined through a mechanism that should be established.
- Low level data design decisions should be deferred until late in design process.
- A library of data structures and operations that are applied to them should be developed.
- The representation of data structure should only be known to those modules that can directly use the data contained within the structure.
- Software design and programming language should support the specification and realization of abstract data types.

Task Analyze what is the basic importance of data design at component level?

Self Assessment

Fill in the blanks:

9. All data structures and …….. to be performed on each should be identified.
10. Software design and programming language should support the specification and realization of …….. data types.

10.3 Architectural Design

As architectural design begins, the design should define the external entities that the software interacts with and nature of the interaction. Once the context is modeled and all external interfaces are explained, the structure of the system is specified by the designer. It is done by defining and refining software components that implement the architecture.

10.3.1 Representing the System in Context

Architectural context represents how the software interacts with entities external to its boundaries. A system context diagram accomplishes this requirement by representing the flow of information into and out of the system. At the architectural design level, a software architect uses an architectural context diagram to model the manner in which software interacts with entities external to its boundaries.

How do systems inter-operate with the target system?

Superordinate Systems: These systems use the target system as part of some higher level processing scheme.

Subordinate Systems: These systems are used by the target system and provide data or processing that are necessary to complete target system.

Peer-level Systems: These systems interact on a peer-to-peer basis.

Actors: These entities interact with the target system by producing or consuming information necessary for requisite processing.

Notes

Caution Each of these external entities communicates with target systems through an interface.

10.3.2 Defining Archetypes

Archetypes are the abstract building blocks of an architectural design. It is a class or pattern that represents a core abstraction that is critical to design of architecture for the target system. Archetypes can be derived by examining analysis classes defined as part of analysis model. Target system architecture is composed of these archetypes which represent stable elements of the architecture. Some kind of archetypes are:

- Nodes
- Detector
- Indicator
- Controller

10.3.3 Refining the Architecture into Components

As the software architecture is refined into components, structure of the system begins to emerge. Components of the software architecture are derived from three sources:

- the application domain
- the infrastructure domain
- the interface domain

Because analysis modeling does not address infrastructure, one should allocate sufficient design time to consider it carefully.

In order to find the components that are most suitable for refining the software architecture, you need to start by using the classes which were explained in the analysis model. The analysis classes in turn are representations of business entities that architecture is trying to explain. You could also base these components on an infrastructure model rather than the business model. If you went purely by the business model, you would not be able to depict many of those infrastructure components such as database components, components used for communication etc.

Notes Whatever interfaces are explained in the architecture context diagram imply one or more specialized components that process the data that flow across the interface.

10.3.4 Describing Instantiations of the System

The context of the system has been represented, the archetypes indicating important abstractions are defined, the overall structure of system apparent and major software components are identified, however further refinement is necessary which is accomplished by developing an actual instantiation of architecture. The architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Self Assessment

Notes

Fill in the blanks:

11. A software architect uses an architectural context diagram to model the manner in which software interacts with …….. to its boundaries.
12. ……. are the abstract building blocks of an architectural design.
13. As architectural design begins, the design should define the external entities that the software interacts with and ……. of the interaction.
14. Target system architecture is composed of these archetypes which represent stable elements of the……...
15. The ……. classes in turn are representations of business entities that architecture is trying to explain.

Caselet

Data Warehousing is next Growth Area: Inmon

MR BILL INMON speaks the language of computers that help better extract information stored in unstructured forms be it speech, e-mail or text, a challenge the industry is trying to find a solution.

Considered as the father of data warehousing, Mr Inmon, is in India speaking about data warehouses and guiding techies in mining data and making them meaningful for enterprises.

Mr Inmon told Business Line data warehouses would grow exponentially; however, the industry is faced with the challenge of extracting unstructured data and making it meaningful, for its users in corporations and Government.

Because of the growing competition, companies cannot do without data warehouse. The data stored in these centres provide the right path for informed decision-making. Areas such as retailing, insurance, banking and telecom services are extremely dependent on data, which make them drill deeper into the existing clients and also offer them right services and solutions.

"Just like the traditional outsourcing services, data warehousing can actually be outsourced to third party depending upon the nature of business. Therefore, I expect a wave of data warehouse outsourcing through the proven global delivery model. However, it is not without own checks and balances," he explained.

Countries such as Germany and Denmark by law prevent any data from leaving the country. "In fact, if you look at large corporations in the US, it is clear that data warehouse is much more entwined with the business itself. SAP, the enterprise software vendor, is one of the world's largest providers of packaged data warehouse and has recently passed 10,000th licence, which is significant. It is here that the issue of business intelligence and analytics comes into play", he said.

Accuracy and integrity of data is a key issue. There are different approaches to data warehouses, one that relates to Oracle and the other that is referred to the IBM way.

Contd...

Notes

Inmon associates: Mr Inmon has created three companies. His first, Prism Solutions, has become part of IBM in 1995 and Pinecomb Systems was sold in 2000.

"I have become old to manage companies. At Inmon Associates Inc based in Colorado, the focus is on data warehouse technologies and addressing its challenges," he said.

10.4 Summary

- Software architecture has started to receive focus since the last decade.
- It is becoming clearer that an early and careful architecture design can greatly reduce the failure rates of software projects.
- A good architecture design partitions the functional requirements of a software system into a manageable set of interacting components.
- Data such as event lists and images need to be kept in files as they are received.
- They also need to be correlated with engineering and other ancillary data which arrive as a continuous time stream and to be associated with a calibration test or an observation ID.
- Once the context is modeled and all external interfaces are explained, the structure of the system is specified by the designer.

10.5 Keywords

Archetypes: Archetypes are the abstract building blocks of an architectural design.

Data warehouse: A data warehouse is a large, independent database that serves the set of applications required by a business.

Data design: Data design translates data objects defined during analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

10.6 Review Questions

1. As the size of software systems increases, the algorithms and data structures of the computation no longer constitute the major design problems. Explain.
2. Data such as event lists and images need to be kept in files as they are received. Discuss.
3. Data design translates data objects defined during analysis model into data structures. Describe.
4. The proposal table contains a record for each proposal which points to the associated file in the file table. Explain how?
5. The content of each data object should be defined through a mechanism that should be established. Analyze.
6. Analyze how architectural design begins; the design should define the external entities that the software interacts with and nature of the interaction.
7. Architectural context represents how the software interacts with entities external to its boundaries. Discuss with an example.
8. Archetypes can be derived by examining analysis classes defined as part of analysis model. Describe.

Notes

9. Explain how the context can be represented in system?
10. Components of the software architecture are derived from which sources?

Answers: Self Assessment

1.	isolation	2.	functional
3.	Metadata	4.	data mining
5.	knowledge-driven	6.	relational
7.	analyze	8.	consistent
9.	operations	10.	abstract
11.	entities external	12.	Archetypes
13.	Nature	14.	Architecture
15.	analysis		

10.7 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering*, Second Edition, PHI.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering a Practioner's Approach*, 5th edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering*, Sixth edition, Pearson Education.

Online links

http://en.wikipedia.org/wiki/Software_architecture

https://www.myarchicad.com/

Notes

Unit 11: Testing Strategies

CONTENTS

Objectives

Introduction

11.1 Software Testing

11.1.1 To Improve Quality

11.1.2 For Verification & Validation (V&V)

11.1.3 History of Software Testing

11.1.4 Problems in Testing

11.1.5 Testing Objectives

11.1.6 Scope of Software Testing

11.2 Testing Process

11.3 Functional Testing

11.4 Structural Testing

11.5 Test Activities

11.5.1 Levels of Testing

11.6 Debugging

11.6.1 Debugging Techniques

11.6.2 Debugging Approaches

11.7 Testing Tools

11.7.1 Testing Strategies for Conventional Software

11.7.2 Testing Strategies for Object Oriented Software

11.8 Summary

11.9 Keywords

11.10 Review Questions

11.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the software testing basics
- Describe the testing process
- Explain the functional testing
- Discuss the structural testing
- Comprehend testing activities

Notes

- Explain debugging
- Scan the testing tools
- Discuss the testing activities for conventional and object oriented software

Introduction

Dear students it should be clear in mind that the philosophy behind testing is to find errors. Test cases are devised with this purpose in mind. A test case is a set of data that the system will process as normal input. However, the data with the express intent of determining whether the system will process then correctly. For example, test cases for inventory handling should include situation in which the quantities to be withdrawn from inventory exceed, equal, and are less than the actual quantities on hand. Each test case is designed with the intent of finding errors in the way the system will process it. There are two general strategies for testing software: code testing and specifications testing. In code testing, the analyst develops that case to execute every instructions and path in a program. Under specification testing, the analyst examines the program specifications and then writes test data to determine how the program operates under specific conditions. Regardless of which strategy the analyst follows, there are preferred practices to ensure that the testing is useful. The levels of tests and types of test data, combined with testing libraries, are important aspects of the actual test process.

11.1 Software Testing

Software Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear — generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects – or bugs – will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable – and humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Discovering the design defects in software is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^64 distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it

Notes

didn't work for previously. But its behavior on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

Caution Regardless of the limitations, testing is an integral part in software development. It is broadly deployed in every phase in the software development cycle.

Typically, more than 50 percent of the development time is spent in testing. Testing is usually performed for the following purposes:

11.1.1 To Improve Quality

As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed space shuttle missions to go awry, halted trading on the stock market, and worse. Bugs can kill. Bugs can cause disasters. The so-called year 2000 (Y2K) bug has given birth to a cottage industry of consultants and programming tools dedicated to making sure the modern world doesn't come to a screeching halt on the first day of the next century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed, is the purpose of debugging in programming phase.

11.1.2 For Verification & Validation (V&V)

Just as topic Verification and Validation indicated, another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We can not test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors – functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. The table below illustrates some of the most frequently cited quality considerations.

Table 11.1: Typical Software Quality Factors

Functionality (exterior quality)	Engineering (interior quality)	Adaptability (future quality)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

Notes

Good testing provides measures for all relevant factors. The importance of any particular factor varies from application to application. Any system where human lives are at stake must place extreme emphasis on reliability and integrity. In the typical business system usability and maintainability are the key factors, while for a one-time scientific program neither may be significant. Our testing, to be fully effective, must be geared to measuring each relevant factor and thus forcing quality to become tangible and visible.

Tests with the purpose of validating the product works are named clean tests, or positive tests. The drawbacks are that it can only validate that the software works for the specified test cases. A finite number of tests can not validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software does not work. Dirty tests, or negative tests, refer to the tests aiming at breaking the software, or showing that it does not work.

Task "A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests." What have you understood from this statement explain with an example.

A testable design is a design that can be easily validated, falsified and maintained. Because testing is a rigorous effort and requires significant time and cost, design for testability is also an important design rule for software development.

11.1.3 History of Software Testing

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979. Although his attention was on breakage testing ("a successful test is one that finds a bug"), it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dr. Dave Gelperin and Dr. William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:

Until 1956 - Debugging oriented

1957-1978 - Demonstration oriented

1979-1982 - Destruction oriented

1983-1987 - Evaluation oriented

1988-2000 - Prevention oriented

Definition

Testing is not a technique for building quality systems; rather, it is a technique that is used because we recognize that we have failed to build a fault free system. Testing assists in its repair by identifying where the program fails to meet its specification. We will broaden the usual definition of testing that typically refers only to the testing of code. The expanded definition includes the testing of the many types of models, requirements, analysis, architectural and detailed design, that are constructed in the early development phases. These models may be represented in an executable format in a CASE tool such as BetterStateÅ or they may require a form of symbolic execution or inspection. My justification for classifying these activities as testing is that the inputs to one of these executions will be much more specific than the inputs to a typical review process.

Notes

Using this expanded definition, there are a number of products of the software development process that should be tested including:

- Requirements models
- Analysis and design models
- Architectural models
- Individual components
- Integrated system code

In fact, there should be a testing activity associated with each step in the development process. Adding the testing of models will find some faults earlier in the development process resulting in cheaper repair costs.

Although these products most often will be tested to determine their correctness, testing may be used to investigate a variety of attributes including:

- Performance
- Usability
- Robustness
- Reusability
- Extensibility
- Flexibility.

The testing perspective is an attitude that questions the validity of every product and utilizes a thorough search of the product to identify faults. Systematic algorithms and intuitive insights guide this search. It is this systematic search that makes testing a more powerful tool than reviews and inspections. A review will almost never find something that isn't there. That is, a review typically only seeks to validate what is in the model and does not systematically search to determine if all the entities and relationships that should be there are. The testing perspective requires that a piece of software demonstrate that it is performing as its complete specification indicates that it should (and that there is no extra behavior). A product is tested to determine that it will do what it is supposed to do (a positive test). It should also be tested to ensure that it does not do what it is not supposed to do (a negative test). This leads to the need to define the parameters of the search.

Test cases are created as part of the testing process to direct the search. A test case presents a context in which the software is to operate and a description of the behavior expected from the software within that context. In most traditional testing situations the context is an operational one in which the software is executed to determine its actual behavior.

Notes If the product under test is a model rather than actual code, the test case may contain a textual description of the context as well as some specific input values.

The testing perspective may be adopted by the same person who developed the product under test or by another person who brings an independent view of the specification and the product. Developer-led testing is efficient since there is no learning curve on the product under test, but it can be less effective because the developer may not search as exhaustively as the independent tester. The completeness and clarity of the specification becomes very important when a second person becomes involved in the testing. If a tester is not given a specification, then any test result is correct!

Notes

11.1.4 Problems in Testing

- Software Testing is a critical element of Software Quality Assurance
- It represents the ultimate review of the requirements, specification, the design and the code.
- It is the most widely used method to ensure Software Quality.
- For most software, exhaustive testing is not possible.
- Many organizations spend 40-50% of development time in testing.
- Testing is the most expensive way to remove defects during software development.
- Problems in testing are one of the major contributors to cost and scheduled overruns.

11.1.5 Testing Objectives

Which ones should we test?

One of the most intense arguments in testing object-oriented systems is whether detailed component testing is worth the effort. That leads me to state an obvious (at least in my mind) axiom: Select a component for testing when the penalty for the component not working is greater than the effort required to test it. Not every class will be sufficiently large, important or complex to meet this test so not every class will be tested independently.

There are several situations in which the individual classes should be tested regardless of their size or complexity:

- ***Reusable components:*** Components intended for reuse should be tested over a wider range of values than a component intended for a single focused use.
- ***Domain components:*** Components that represent significant domain concepts should be tested both for correctness and for the faithfulness of the representation.
- ***Commercial components:*** Components that will be sold, as individual products should be tested not only as reusable components but also as potential sources of liability.

11.1.6 Scope of Software Testing

A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the quality aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members.

Did u know? **What is the role of information derived from the software testing?**

Information derived from software testing may be used to correct the process by which software is developed.

Notes

Self Assessment

Fill in the blanks:

1. Unlike most physical systems, most of the defects in software are design errors, not …….. defects.
2. …… means the conformance to the specified design requirement.
3. …….., a narrow view of software testing, is performed heavily to find out design defects by the programmer.
4. Test cases are created as part of the …….. process to direct the search.
5. Testing is the most expensive way to remove defects during software……..
6. Discovering the design defects in software is equally difficult, for the same reason of …….

11.2 Testing Process

A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process).

Testing can be done on the following levels:

- Unit testing tests the minimal software component, or module. Each unit (basic component) of the software is tested to verify that the detailed design for the unit has been correctly implemented. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.
- Integration testing exposes defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.
- System testing tests a completely integrated system to verify that it meets its requirements. System integration testing verifies that a system is integrated to any external or third party systems defined in the system requirements.

Before shipping the final version of software, alpha and beta testing are often done additionally:

- Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Notes

- Beta testing comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Finally, acceptance testing can be conducted by the end-user, customer, or client to validate whether or not to accept the product. Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Task Visit a software testing website and analyze the limitations of alpha and beta testing.

Regression Testing

After modifying software, either for a change in functionality or to fix defects, a regression test re-runs previously passing tests on the modified software to ensure that the modifications haven't unintentionally caused a regression of previous functionality. Regression testing can be performed at any or all of the above test levels. These regression tests are often automated.

More specific forms of regression testing are known as sanity testing, when quickly checking for bizarre behavior, and smoke testing when testing for basic functionality.

Benchmarks may be employed during regression testing to ensure that the performance of the newly modified software will be at least as acceptable as the earlier version or, in the case of code optimization, that some real improvement has been achieved.

Finding Faults

It is commonly believed that the earlier a defect is found the cheaper it is to fix it. The following table shows the cost of fixing the defect depending on the stage it was found. For example, if a problem in requirements is found only post-release, then it would cost 10-100 times more to fix it comparing to the cost if the same fault was already found by the requirements review.

Time Introduced	Time Detected				
	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

Self Assessment

Fill in the blanks:

7. Unit testing tests the …… software component, or module.
8. ……. testing verifies that a system is integrated to any external or third party systems defined in the system requirements.
9. Alpha testing is simulated or actual operational testing by potential users/customers or an ……. test team at the developers' site.
10. It is commonly believed that the earlier a defect is found the …….. it is to fix it.

Notes

11.3 Functional Testing

Software testing methods are traditionally divided into black box testing and white box testing. Functional testing is also known as black box testing. Black box testing treats the software as a black-box without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing, specification based testing, etc. Black-box test design treats the system as a "black-box", so it doesn't explicitly use knowledge of the internal structure. Black-box test design is usually described as focusing on testing functional requirements.

Did u know? **What are the synonyms of Black-box testing?**

Synonyms for black-box include: behavioral, functional, opaque-box, and closed-box.

Specification based Testing

Specification based Testing aims to test the functionality according to the requirements. Thus, the tester inputs data and only sees the output from the test object. This level of testing usually requires thorough test cases to be provided to the tester who then can simply verify that for a given input, the output value (or behavior), is the same as the expected value specified in the test case. Specification based testing is necessary but insufficient to guard against certain risks.

Non-functional Software Testing

Special methods exist to test non-functional aspects of software.

Performance testing checks to see if the software can handle large quantities of data or users. This is generally referred to as software scalability.

Usability testing is needed to check if the user interface is easy to use and understand.

Security testing is essential for software which processes confidential data and to prevent system intrusion by hackers.

Internationalization and localization is needed to test these aspects of software, for which a pseudo-localization method can be used.

Self Assessment

Fill in the blank:

11. Black box testing treats the software as a black-box without any knowledge of ……. implementation.

11.4 Structural Testing

Structural Testing is also known as the white box testing. White box testing, by contrast to black box testing, is when the tester has access to the internal data structures and algorithms (and the code that implement these). While black-box and white-box are terms that are still in popular use, many people prefer the terms "behavioral" and "structural". Behavioral test design is slightly different from black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged. In practice, it hasn't proven useful to use a single test design

Notes

method. One has to use a mixture of different methods so that they aren't hindered by the limitations of a particular one. Some call this "gray-box" or "translucent-box" test design, but others wish we'd stop talking about boxes altogether. White-box test design allows one to peek inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data.

Did u know? **What are synonyms of White box testing?**

Synonyms for white-box include: structural, glass-box and clear-box.

Types of White Box Testing

The following types of white box testing exist:

Code coverage: creating tests to satisfy some criteria of code coverage. For example, the test designer can create tests to cause all statements in the program to be executed at least once.

Mutation testing methods

Fault injection methods.

Static testing: White box testing includes all static testing.

Code Completeness Evaluation

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.

Two common forms of code coverage are: function coverage, which reports on functions executed and statement coverage, which reports on the number of lines executed to complete the test.

They both return coverage metric, measured as a percentage.

Grey Box Testing

In recent years the term grey box testing has come into common usage. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level.

Manipulating input data and formatting output do not qualify as grey-box because the input and output are clearly outside of the black-box we are calling the software under test. This is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test.

Notes Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

Notes

Self Assessment

Fill in the blank:

12. ……test design allows one to peek inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data.

11.5 Test Activities

- ***Compatibility Test:*** Test to ensure compatibility of an application or Web site with different browsers, OS, and hardware platforms. Compatibility testing can be performed manually or can be driven by an automated functional or regression test suite.
- ***Conformance Test:*** Verifying implementation conformance to industry standards. Producing tests for the behavior of an implementation to be sure it provides the portability, interoperability, and/or compatibility a standard defines.
- ***Functional Test:*** Validating an application or Web site conforms to its specifications and correctly performs all its required functions. This entails a series of tests, which perform a feature-by-feature validation of behavior, using a wide range of normal and erroneous input data. This can involve testing of the product's user interface, APIs, database management, security, installation, networking; etc testing can be performed on an automated or manual basis using black box or white box methodologies.
- ***Load Test:*** Load testing is a generic term covering Performance Testing and Stress Testing.
- ***Performance Test:*** Performance testing can be applied to understand your application or WWW site's scalability, or to benchmark the performance in an environment of third party products such as servers and middleware for potential purchase. This sort of testing is particularly useful to identify performance bottlenecks in high use applications. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions.
- ***Regression Test:*** Similar in scope to a functional test, a regression test allows a consistent, repeatable validation of each new release of a product or Web site. Such testing ensures reported product defects have been corrected for each new release and that no new quality problems were introduced in the maintenance process. Though regression testing can be performed manually an automated test suite is often used to reduce the time and resources needed to perform the required testing.
- ***Smoke Test:*** A quick-and-dirty test that the major functions of a piece of software work without bothering with finer details. Originated in the hardware testing practice of turning on a new piece of hardware for the first time and considering it a success if it does not catch on fire.
- ***Stress Test:*** Test conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how. A graceful degradation under load leading to non-catastrophic failure is the desired result.

Caution Often Stress Testing is performed using the same process as Performance Testing but employing a very high level of simulated load.

Notes

11.5.1 Levels of Testing

Unit Testing

In unit testing, the analyst tests the program making up a system. For this reason, unit testing is sometimes called program testing. Unit testing gives stress on the modules independently of one another, to find errors. This helps the tester in detecting errors in coding and logic that are contained within that module alone. The errors resulting from the interaction between modules are initially avoided. For example, a hotel information system consists of modules to handle reservations; guest check in and checkout; restaurant, room service and miscellaneous charges; convention activities; and accounts receivable billing. For each, it provides the ability to enter, modify or retrieve data and respond to different types of enquiries or print reports. The test cases needed for unit testing should exercise each condition and option.

Unit testing can be performed from the bottom up, starting with smallest and lowest level modules and proceedings one at a time. For each module in bottom-up testing a short program is used to execute the modules and provided the needed data, so that the modules is asked to perform the way it will when embedded within the larger system.

System Testing

The important and essential part of the system development phase, after designing and developing the software is system testing. We cannot say that every program or system design is perfect and because of lack of communication between the user and designer, some error is there in the software development. The number and nature of errors in a newly designed system depend on some usual factors like communication between the user and the designer; the programmer's ability to generate a code that reflects exactly the systems specifications and the time frame for the design.

Theoretically, a newly designed system should have all the parts or sub-systems are in working order, but in reality, each sub-system works independently. This is the time to gather all the sub-system into one pool and test the whole system to determine whether it meets the user requirements. This is the last change to detect and correct errors before the system is installed for user acceptance testing. The purpose of system testing is to consider all likely variations to which it will be subjected and then push the system to its limits.

Testing is an important function of the success of the system. System testing makes a logical assumption that if all the parts of the system are correct, the goal will be successfully activated.

Notes Another reason for system testing is its utility as a user-oriented vehicle before implementation.

System Testing consists of the following five steps:

1. ***Program Testing:*** A program represents the logical elements of a system. For a program to run satisfactorily, it must compile and test data correctly and tie in properly with other programs. It is the responsibility of a programmer to have an error free program. At the time of testing the system, there exist two types of errors that should be checked. These errors are syntax and logical. A syntax error is a program statement that violates one or more rules of the language in which it is written. An improperly defined field dimension or omitted key words are common syntactical errors. A logical error, on the other hand, deals with incorrect data fields out of range items, and invalid combinations. Compiler

Notes

like syntax errors does not detect them; the programmer must examine the output carefully to detect them.

2. ***String Testing:*** Programs are invariably related to one another and interact in a total system. Each program is tested to see whether it conforms to related programs in the system. Each part of the system is tested against the entire module with both test and live data before the whole system is ready to be tested.

3. ***System Testing:*** It is deigned to uncover weaknesses that were not found in earlier tests. This includes forced system failure and validation of total system, as its user in the operational environment will implement it. Under this testing, we take low volumes of transactions based on live data. This volume is increased until the maximum level for each transaction type is reached.

4. ***System Documentation:*** All design and test documentation should be well prepared and kept in the library for future reference. The library is the central location for maintenance of the new system.

5. ***User Acceptance Testing:*** Acceptance test has the objective of selling the user on the validity and reliability of the system. It verifies that the system's procedures operate to system specifications and that the integrity of important data is maintained. Performance of an acceptance test is actually the user's show. User motivation is very important for the successful performance of the system.

Self Assessment

Fill in the blanks:

13. The errors resulting from the interaction between ..... are initially avoided.

14. The library is the central location for ..... of the new system.

11.6 Debugging

The goal of testing is to identify errors in the program. The process of testing gives symptoms of the presence of an error. After getting the symptom, we begin to investigate to localize the error i.e. to find out the module causing the error. This section of code is then studied to find the cause of the problem. This process is called debugging. Hence, debugging is the activity of locating and correcting errors. It starts once a failure is detected.

11.6.1 Debugging Techniques

Although developers learn a lot about debugging from their experiences, but these are applied in a trial and error manner. Debugging is not an easy process as per human psychology as error removal requires the acceptance of error and an open mind willing to admit the error.

However, Pressman explains certain characteristics of bugs that can be useful for developing a systematic approach towards debugging. These are:

The symptoms and cause may be geographically remote. That is, the symptom may appear in one part of the program, while the cause may actually be located in other part. Highly coupled programs can worsen this situation.

The symptom may disappear (temporarily) when another error is corrected.

The symptom may actually be caused by non errors (e.g. round off inaccuracies).

Notes

The symptom may be caused by a human error that is not easily traced.

The symptom may be result of timing problems rather than processing problems.

It may be difficult to accurately reproduce input conditions (e.g. a real time application in which input ordering is indeterminate).

The symptom may be intermittent. This is particularly common in embedded systems that couple hardware with software inextricably.

The symptom may be due to causes that are distributed across a number of tasks running on different processors.

11.6.2 Debugging Approaches

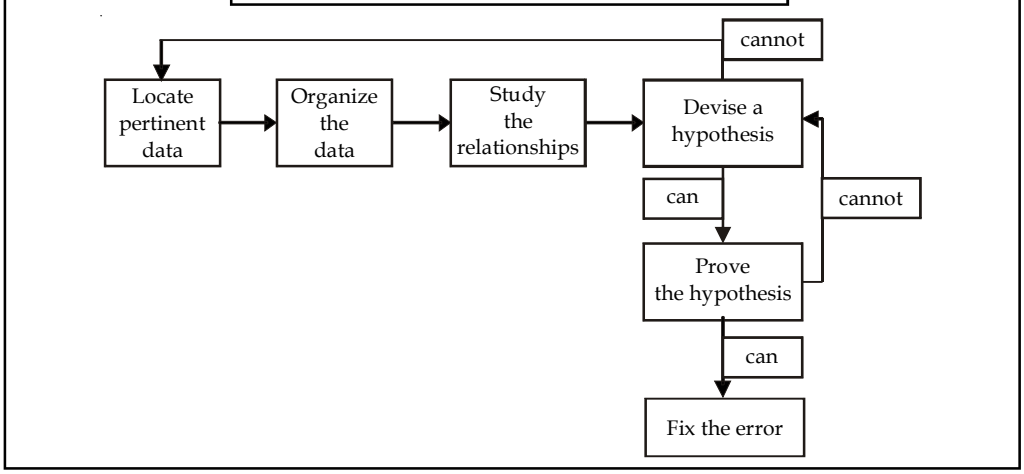
The debugging approach can be categorized into various categories. The first one is trial and error. The debugger looks at the symptoms of the errors and tries to figure out that from exactly which part of the code the error originated. Once found the cause, the developer fixes it. However, this approach is very slow and a lot of time and effort goes waste.

The other approach is called backtracking. Backtracking means to examine the error symptoms to see where they were observed first. One then backtracks in the program flow of control to a point where these symptoms have disappeared. This process identifies the range of the program code which might contain the cause of errors. Another variant of backtracking is forward tracking, where print statements or other means are used to examine a sequence of intermediate results to determine the point at which the result first becomes wrong.

The third approach is to insert watch points (output statements) at the appropriate places in the program. This can be done using software without manually modifying the code.

The fourth approach is more general and called induction and deduction. The induction approach comes from the formulation of a single working hypothesis based on the data, analysis of the existing data and on especially collected data to prove or disprove the working hypothesis. The inductive approach is explained in Figure 11.1.

Figure 11.1: The Inductive Debugging Approach

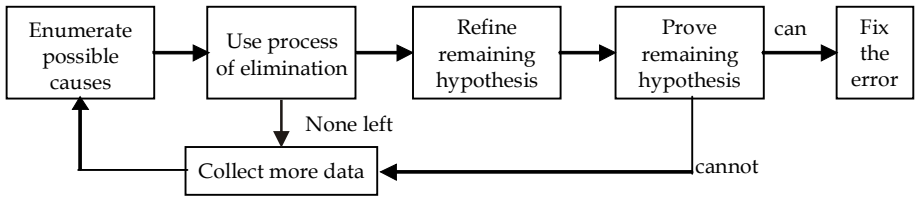


Task Analyze the various categories in which the debugging approaches can be categorized.

Notes

The deduction approach begins by enumerating all causes or hypothesis, which seem possible. Then, one by one, particular causes are ruled out until a single one remains for validation. This is represented in Figure 11.2.

Figure 11.2: The Deductive Debugging Approach



Self Assessment

Fill in the blank:

15. Debugging is the activity of locating and …….. errors.

11.7 Testing Tools

Program testing and fault detection can be aided significantly by Testing tools and debuggers. Types of testing/debug tools include features such as:

1. Program monitors, permitting full or partial monitoring of program code including: Instruction Set Simulator, permitting complete instruction level monitoring and trace facilities.
2. Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code or code coverage reports.
3. Formatted dump or Symbolic debugging, tools allowing inspection of program variables on error or at chosen points.
4. Benchmarks, allowing run-time performance comparisons to be made.
5. Performance analysis or profiling tools that can help to highlight hot spots and resource usage.

Notes Some of these features may be incorporated into an integrated development environment (IDE).

11.7.1 Testing Strategies for Conventional Software

The various testing strategies for conventional software.

- Unit testing
- Integration testing
- Validation testing
- System testing

Unit testing, Integration Testing and System Testing have been explained in this unit.

Notes

Validation Testing

Software validation testing is essentially a task carried out by a software tester. The aim is to check, if the software has been made in lines with the requirements of the client. In this article, we will talk about software validation testing in detail.

While verification is a quality control process, quality assurance process carried out before the software is ready for release is known as validation testing. The validation testing goals is to validate and be confident about the software product or system, that it fulfills the requirements given by the customer. The acceptance of the software from the end customer is also a part of validation testing.

Validation testing answers the question, "Are you building the right software system". Another question, which the entire process of validation testing in software engineering answers is, "Is the deliverable fit for purpose". In other words, does the software system provide the right solution to the problem. Therefore, often the testing activities are introduced early in the software development life cycle.

Notes The two major areas, when validation testing should take place are in the early stages of software development and towards the end, when the product is ready for release. In other words, it is acceptance testing which is a part of validation testing.

Validation Testing Types

If the testers are involved in the software product right from the very beginning, then validation testing in software testing starts right after a component of the system has been developed. The different types of software validation testing are:

1. ***Component Testing:*** Component testing is also known as unit testing. The aim of the tests carried out in this testing type is to search for defects in the software component. At the same time, it also verifies the functioning of the different software components, like modules, objects, classes, etc., which can be tested separately.
2. ***Integration Testing:*** This is an important part of the software validation model, where the interaction between the different interfaces of the components is tested. Along with the interaction between the different parts of the system, the interaction of the system with the computer operating system, file system, hardware and any other software system it might interact with is also tested.
3. ***System Testing:*** System testing, also known as functional and system testing is carried out when the entire software system is ready. The concern of this testing is to check the behavior of the whole system as defined by the scope of the project. The main concern of system testing is to verify the system against the specified requirements. While carrying out the tester is not concerned with the internals of the system, but checks if the system behaves as per expectations.
4. ***Acceptance Testing:*** Here the tester especially has to literally think like the client and test the software with respect to user needs, requirements and business processes and determine, whether the software can be handed over to the client. At this stage, often a client representative is also a part of the testing team, so that the client has confidence in the system. There are different types of acceptance testing:
   - Operational Acceptance Testing

Notes

- Compliance Acceptance Testing
- Alpha Testing
- Beta Testing

Often when validation testing interview questions are asked, they revolve around the different types of validation testing. The difference between verification and validation is also a common software validation testing question. Some organizations may use different terms for some of the terms given in the article above. As far as possible, we have tried to accept the alternate names as well.

11.7.2 Testing Strategies for Object Oriented Software

In testing object oriented software, the objective of testing remains the same. The nature of object oriented software changes both testing strategy and testing tactics.

Unit Testing in Object Oriented Context

In object oriented software, an encapsulated class is the focus of unit testing. Encapsulation means that class and class object packages the attributes and operations that manipulate the data. A class contains many operations so unit testing these operations must also vary.

Class testing for object oriented software is analogous to module testing for conventional software. It is not advisable to test operations in isolation. Unit testing of conventional software focus on algorithmic detail of module and the data that flows across module whereas unit testing for object oriented software focus on operations encapsulated by class.

Integration Testing in Object Oriented Context

An important strategy for integration testing of object oriented software is thread based testing. Threads are set of classes that respond to an input or event. Each thread is integrated and tested individually. Regression testing is done.

Another strategy is the use based tests that focus on classes that do not collaborate heavily with other classes.

The behavior of drivers and stubs also changes during integration testing of object oriented software. Drivers test operations at lowest level and for testing whole group of classes.

Did u know?

What are the uses of stubs?

Stubs are used where collaboration between classes is required but one or more collaborating class is not fully implemented.

Self Assessment

Fill in the blanks:

1. …….. analysis or profiling tools that can help to highlight hot spots and resource usage.
2. ……. for object oriented software is analogous to module testing for conventional software.

Notes

Caselet

Kshema Plans Software Testing, Validation Services

KSHEMA Technologies, a Bangalore-based software services firm, as part of its diversification, is planning to offer software testing and validation services.

The company recently set up a team of five engineers in this regard, which would be expanded later, said Mr Anant Koppar, CEO, Kshema Technologies.

"The need to set up a separate testing and validation team was primarily on the client's insistence," Mr Koppar said, adding that the company would also offer third party software testing services and certification later.

Two of our major customers had shown interest in availing of these services, he added.

The testing team would be part of the company's engineering services division, set up recently as part of the organisational restructuring.

Through the division, the company not only plans to provide end-to-end solutions in engineering design services, but also intends to offer quality consultancy.

The company had a team of four lead auditors certified by BVQI, Mr Koppar said, adding that it was currently handling three to four consultancy assignments from the existing customers.

In order to strengthen its engineering services division, Kshema is also planning to tie up with three/ four small partners for design development. These partners would be essentially small-time engineering companies and the company expected to announce the partnerships in the next few weeks, he added.

Mr Koppar also said that the company expected to achieve the SEI CMM Level 5 by June this year.

The company, which has successfully developed and implemented several technologies on the Microsoft .Net platform, was in talks with Microsoft to get itself declared as a .Net Competency Centre, he said.

The industry slowdown had impacted the company's growth, which is likely to be between 10-20 per cent for the current year.

The company recorded a turnover of ₹ 57 crore in the last financial year.

11.8 Summary

- Software testing is the lifeline of any successful Project Implementation.
- Right tools and methods help in correctly identifying problems and correcting them before live runs.
- Software Testing is the process of executing a program or system with the intent of finding errors.
- Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.
- A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer.

Notes

- Software testing methods are traditionally divided into black box testing and white box testing. Functional testing is also known as black box testing.
- Structural Testing is also known as the white box testing. White box testing, by contrast to black box testing, is when the tester has access to the internal data structures and algorithms.
- The goal of testing is to identify errors in the program.
- The process of testing gives symptoms of the presence of an error.

11.9 Keywords

Debugging: Debugging is the activity of locating and correcting errors. It starts once a failure is detected.

ET: Error Tolerance

Functional Testing: Functional testing is also known as black box testing.

IDE: Integrated Development Environment

Software Testing: Software Testing is the process of executing a program or system with the intent of finding errors.

Structural Testing: Structural Testing is also known as the white box testing.

11.10 Review Questions

1. Software Testing is the process of executing a program or system with the intent of finding errors. Analyze.
2. What do you know about various special system tests? Explain briefly.
3. Discuss the various debugging approaches.
4. Explain how Software bugs will almost always exist in any software module with moderate size. Give examples.
5. Discuss how Software Testing is a critical element of Software Quality Assurance. Give reasons.
6. How System testing tests a completely integrated system to verify that it meets its requirements.
7. Specification Based Testing aims to test the functionality according to the requirements. Analyze this statement.
8. Behavioral test design is slightly different from black-box test design. Comment.
9. Testing is an important function of the success of the system. Discuss.
10. Make the distinction between verification and validation. Explain with example.

Answers: Self Assessment

1.	manufacturing	2.	Quality
3.	Debugging	4.	testing
5.	development	6.	complexity

Notes

7. minimal
8. System integration
9. Independent
10. Cheaper
11. internal
12. White-box
13. modules
14. maintenance
15. correcting
16. Performance
17. Class testing

11.11 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering*, Second Edition, PHI.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering a Practioner's Approach*, 5th edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering*, Sixth edition, Pearson Education.

Online links

http://msdn.microsoft.com/en-us/library/ms978235.aspx

http://bazman.tripod.com/what_testing.html

Notes

Unit 12: Testing Tactics

CONTENTS

Objectives

Introduction

12.1 White-box testing

12.1.1 White-box Testing Techniques

12.2 Black-Box Testing

12.2.1 Black-box Testing Techniques

12.3 Fault-based Testing

12.4 Summary

12.5 Keywords

12.6 Review Questions

12.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Describe black-box testing
- Recognize white-box testing
- Scan flow graph testing
- Explain equivalence partitioning
- Comprehend boundary value analysis
- Examine Fault based testing

Introduction

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software: we can not completely test a program with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality.

12.1 White-Box testing

White-box testing or glass-box testing is a test case design method that uses the control structure of the procedural design to obtain test cases. Using this methodology, a software engineer can come up with test cases that guarantee that all independent paths within a module have been

exercised at least once; exercise all the logical decisions based on their true and false sides; (3) executing all loops at their boundaries and within them; and exercise internal data structures to ensure their validity.

The reason of conducting white-box testing largely depends upon the nature of defects in software:

- Logic 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 likely to be executed when, in fact, it may be executed on a regular basis.
- Typographical errors are random.

Contrary to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called glass-box testing, logic-driven testing or design-based testing.

Task Analyze why white box testing is known as glass box testing?

There are many techniques available in white-box testing, because the problem of intractability is eased by specific knowledge and attention on the structure of the software under test. The intention of exhausting some aspect of the software is still strong in white-box testing, and some degree of exhaustion can be achieved, such as executing each line of code at least once (statement coverage), traverse every branch statements (branch coverage), or cover all the possible combinations of true and false condition predicates (Multiple condition coverage).

Control-flow testing, loop testing, and data-flow testing, all maps the corresponding flow structure of the software into a directed graph. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code — code that is of no use, or never get executed at all, which can not be discovered by functional testing.

In mutation testing, the original program code is perturbed and many mutated programs are created, each contains one fault. Each faulty version of the program is called a mutant. Test data are selected based on the effectiveness of failing the mutants. The more mutants a test case can kill, the better the test case is considered. The problem with mutation testing is that it is too computationally expensive to use. The boundary between black-box approach and white-box approach is not clear-cut. Many testing strategies mentioned above, may not be safely classified into black-box testing or white-box testing. It is also true for transaction-flow testing, syntax testing, finite-state testing, and many other testing strategies not discussed in this text. One reason is that all the above techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad — it may contain any requirement including the structure, programming language, and programming style as part of the specification content.

We may be reluctant to consider random testing as a testing technique. The test case selection is simple and straightforward: they are randomly chosen. Some very subtle errors can be discovered with low cost. And it is also not inferior in coverage than other carefully designed testing techniques. One can also obtain reliability estimate using random testing results based on operational profiles.

Notes

Caution Effectively combining random testing with other testing techniques may yield more powerful and cost-effective testing strategies.

12.1.1 White-box Testing Techniques

White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases.

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

1. guarantee that all independent paths within a module have been exercised at least once
2. exercise all logical decisions on their true and false sides
3. execute all loops at their boundaries and within their operational bounds, and
4. exercise internal data structures to ensure their validity.

Basis Path Testing

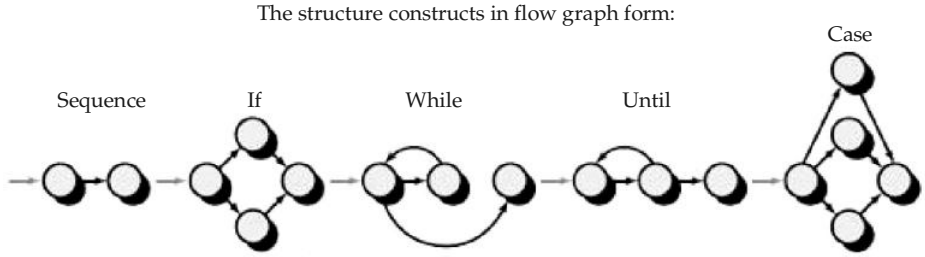
Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow Graph Notation

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

Figure 12.1

The structure constructs in flow graph form:



Where each circle represents one or more non-branching PDL or source code statements

Referring to the figure, each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.

Notes

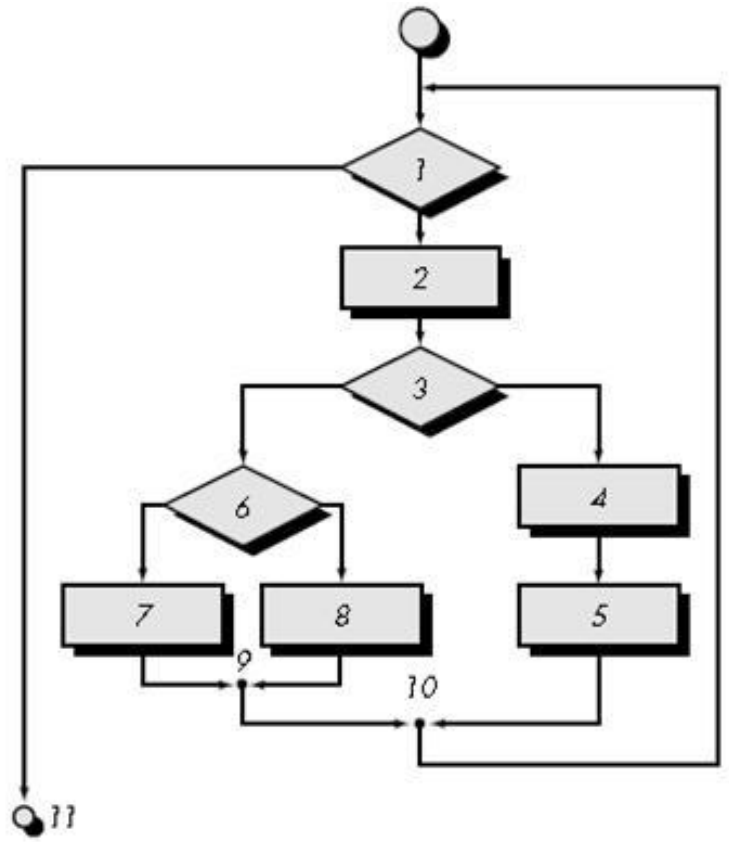
Notes Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

Cyclomatic Complexity/Independent Program Path

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

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

Figure 12.2 (A)



Notes

Figure 12.2 (B)

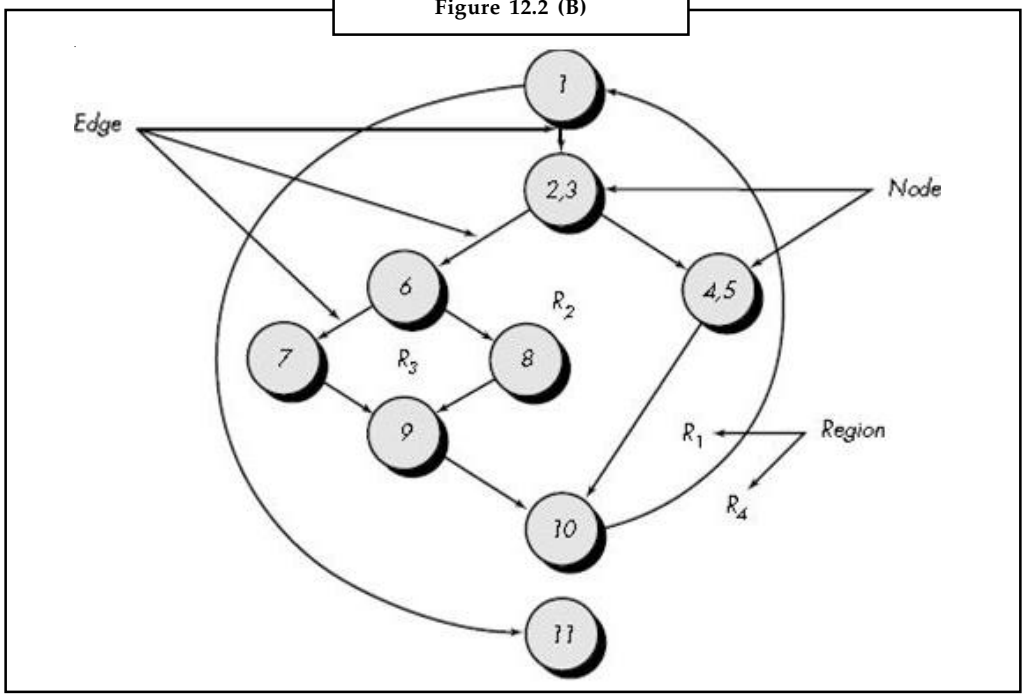
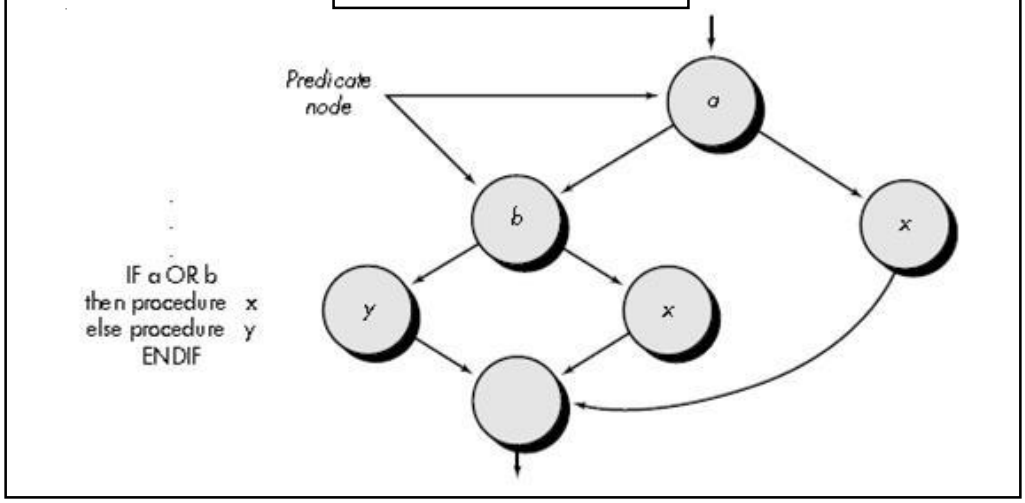


Figure 12.2 (C)



Example: A set of independent paths for the flow graph illustrated in Figure 12.2 (B) is:

path 1: 1-11

path 2: 1-2-3-4-5-10-1-11

path 3: 1-2-3-6-8-9-10-1-11

path 4: 1-2-3-6-7-9-10-1-11

Notes

Notes Note that each new path introduces a new edge.

The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the flow graph in Figure 12.2 (B).

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

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

1. The number of regions of the flow graph correspond to the cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E – N + 2 where E is the number of flow graph edges, N is the number of flow graph nodes.
3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P + 1 where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 12.2 (B), the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. V(G) = 11 edges – 9 nodes + 2 = 4.
3. V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 12.2 (B) is 4.

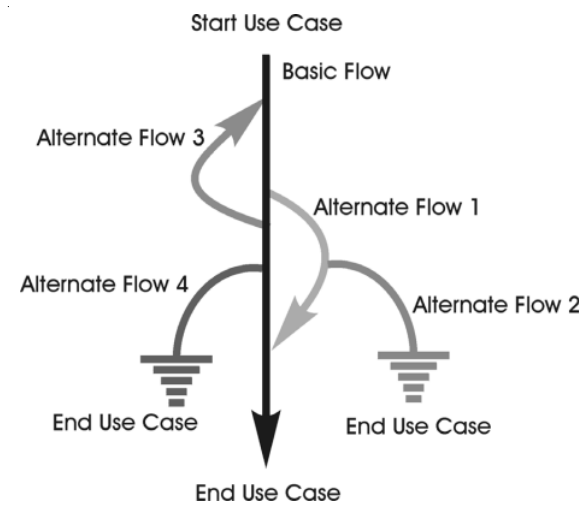
More important, the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Deriving Test Cases

1. Using the design or code as a foundation, draw a corresponding flow graph. A flow graph is created using the symbols and construction rules. Referring to the PDL for average in Figure 12.3 (A), a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 12.3 (B).

Notes

Figure 12.3 (A)



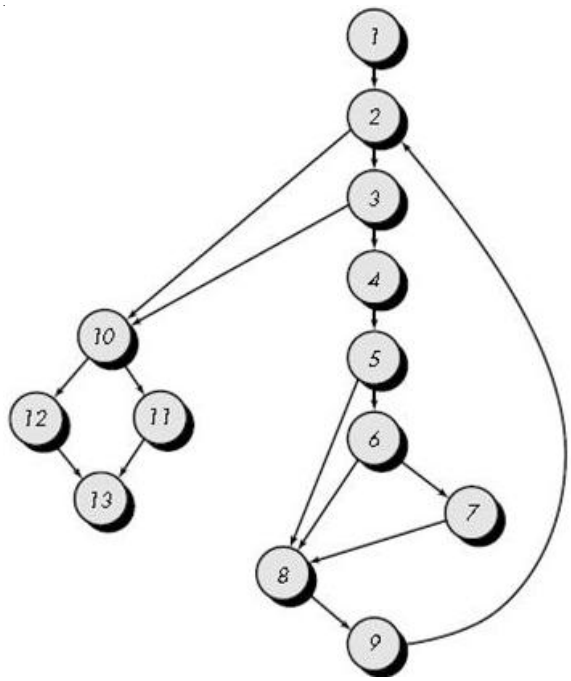
2. Determine the cyclomatic complexity of the resultant flow graph. The cyclomatic complexity, V(G), is determined. It should be noted that V(G) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1. Referring to Figure 12.3 (B),

   V(G) = 6 regions

   V(G) = 17 edges – 13 nodes + 2 = 6

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

Figure 12.3 (B)



Notes

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

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

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

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

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

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

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

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

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

   Path 1 test case: value(k) = valid input, where k < i for 2 = i = 100

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

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

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

   Path 2 test case: value(1) = -999

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

   Path 3 test case: Attempt to process 101 or more values. First 100 values should be valid.

   Expected results: Same as test case 1.

   Path 4 test case: value(i) = valid input where i < 100

   value(k) < minimum where k < i

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

   Path 5 test case: value(i) = valid input where i < 100

   value(k) > maximum where k <= i

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

   Path 6 test case: value(i) = valid input where i < 100

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

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

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

Notes

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

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

Self Assessment

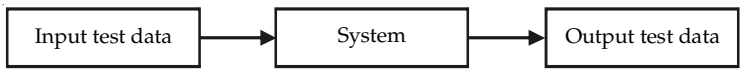
Fill in the blanks:

1. Test cases are derived from the …….. structure.
2. An independent path is any path through the program that introduces at least one new set of processing …….. or a new condition.
3. White box testing can test paths within a unit, paths between units during……., and between subsystems during a system level test.
4. …… complexity is software metric that provides a quantitative measure of the logical complexity of a program.
5. Logic errors and incorrect assumptions are ……. proportional to the probability that a program path will be executed.
6. Testing plans are made according to the details of the software……...
7. While white-box testing can be applied at the unit, integration and …….. of the software testing process, it is usually done at the unit level.

12.2 Black-box Testing

Black-box testing or behavioral testing, mainly focuses on the functional requirements of the software. It enables the software engineer to develop a set of inputs that can fully involve all the functional requirements for a program. It is not an alternative approach to white-box testing. Rather, it completes the testing by uncovering different types of errors than the white-box approach.

Figure 12.4: Black-box Testing



This approach attempts to find errors in the following categories:

1. incorrect or missing functions,
2. interface related errors,
3. errors in data structures or database accesses,
4. performance related errors and
5. initialization and termination errors.

Black box testing is a software testing techniques in which functionality of the software under test (SUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software.

Notes

Did u know? **What are Tools used for Black-box Testing?**

Tools used for Black-box testing largely depends on the type of black box testing your are doing.

- For Functional/Regression Tests you can use – QTP
- For Non-Functional Tests you can use – Loadrunner

This technique is applied at a later stage unlikely to the white-box testing approach. Using this technique, we can arrive at test cases that satisfy the following criteria:

1. test cases that reduce the total test cases by a large quantity to achieve reasonable testing and
2. test cases that tell us something about the presence or absence of a variety of errors rather than a particular kind of error.

The techniques are as below.

Boundary Value Analysis

The art of testing is to come up with a small set of test cases such that the chances of detecting an error are maximized while minimizing the chances of creating redundant test cases that uncover similar errors. It has been observed that the probability of finding errors increases if the test cases are designed to check boundary values.

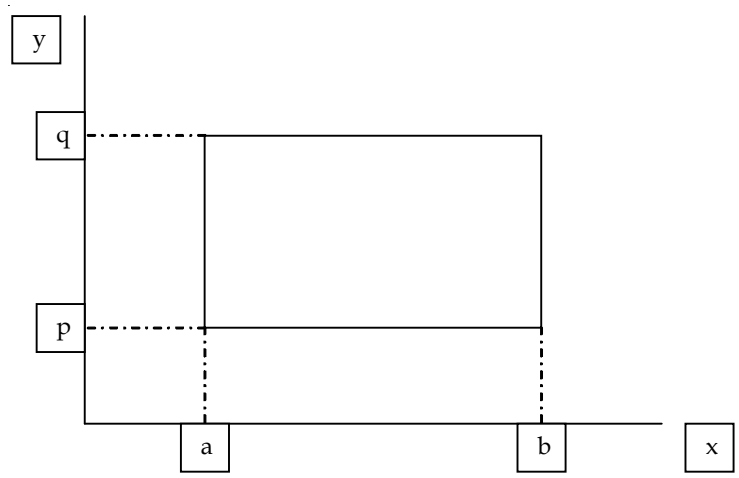
Consider a function F with x and y as two input variables. These inputs will have some boundaries:

$$a<=x<=b$$

$$p<=y<=q$$

Hence, inputs x and y are bounded by two intervals [a,b] and [p,q] respectively. For x, we can design test cases with values a and b, just above a and b and just below a and b which will have higher chances to detect errors. Similar is the case for y. This is represented in the Figure 12.5.

Figure 12.5: Input Domain of a Function of Two Variables



Boundary value test cases are obtained by keeping one variable at its extreme and the other at its nominal value. At times we might want to check the behavior when the value of one variable

Notes

exceeds its maximum. This is called robustness testing. We may also include test cases to check what happens when more than one variable have maximum values. This is called worst case analysis.

Did u know? **What is ad-hoc testing?**

Using prior testing knowledge and experience to test similar programs is called ad-hoc testing.

Equivalence Class Testing

In this method of testing, input domain of a program is divided into a finite number of equivalence classes such that the test of a representative value of each class is equivalent to a test of any other value. That is if a test in a class detects one error all other test cases belonging to the same class must detect the same error. Also, if a test case in a class did not detect an error the other test cases of the same class also should not detect the error. This method of testing is implemented using the following two steps:

1. The equivalence class is identified by taking each input condition and dividing it into valid and invalid classes.
2. Developing test cases using the classes identified. This is done by writing test cases covering all the valid equivalence classes and a single case for invalid equivalence class.

Again good test cases will be those that check for boundary value condition.

Decision Table based Testing

Decision tables are useful for describing situations in which a number of combinations of actions are taken under varying sets of conditions. There are four parts of a decision table namely, Condition stub, Action stub, Condition entries and Action entries. These are described in Figure 12.6.

Figure 12.6: Decision Table Terminology

Condition Stub	Entry						
C_1	True				False		
C_2	True		False		True		False
C_3	True	False	True	False	True	False	-
Action Stub							
A_1	X	X			X		
A_2	X		X			X	
A_3		X			X		
A_4				X		X	X

To develop test cases from decision tables, we treat conditions as input and actions as outputs. Sometimes conditions end up referring to equivalence classes of inputs, and actions refers to major functional processing portions of the item being tested.

Cause Effect Graphing Technique

One drawback of boundary value analysis and equivalence partitioning is that these they do not explore combinations of input circumstances which may result in interesting conditions. These

Notes

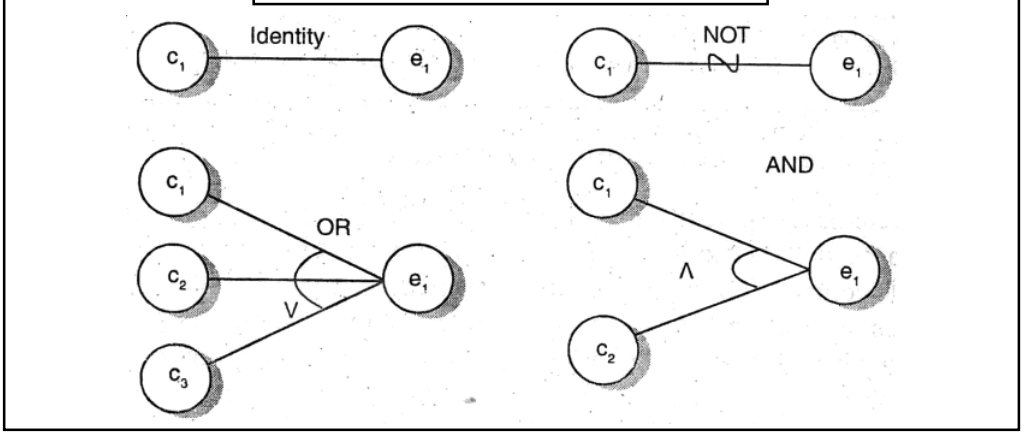
situations must be tested. If we consider all possible valid combinations of equivalence classes, then it results in a large number of test cases, many of which may not uncover any undiscovered errors.

This technique helps in selecting, in a systematic approach, a high-yield set of test cases. It is also useful in pointing out incompleteness and ambiguities in the specifications. The following steps are used to derive test cases:

1. ***The causes and effects are identified.*** A cause is a distinct input condition and effects are output conditions or a system transformation. These are identified by reading the specification and identifying the words or phrases that describe causes and effects. Each cause and effect is assigned a unique number.
2. The semantic content of specification is studied and transformed into a Boolean graph linking the causes and effects. This is the cause effect graph.
3. The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.
4. By methodically tracing state conditions in the graph, the graph is converted into a limited entry decision table. Each column in the graph represents a test case.

The columns in the decision table are converted into test cases. The basic notation for the graph is shown in Figure 12.7.

Figure 12.7: Basic Cause Effect Graph Symbols



Think of each node as having values either 0 or 1, where 0 represents the 'absent state' and 1 represents the 'present state'. The identity function states that if c_1 is 1, e_1 is 1; else e1 is 0. The NOT function states that if c_1 is 1, e_1 is 0 and *vice versa*. The OR function states that if c_1 or c_2 or c_3 is 1, e_1 is 1 else e_1 is 0. The AND function states that if both c_1 and c_2 are 1, e_1 is 1 else e_1 is 0. The AND and OR functions can have any number of inputs.

The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure. It is also termed data-driven, input/output driven or requirements-based testing. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing — a testing method emphasized on executing the functions and examination of their input and output data. The tester treats the software under test as a black box — only the inputs, outputs and specification are visible, and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification.

Notes

Task No implementation details of the code are considered. Analyze this statement.

It is obvious that the more we have covered in the input space, the more problems we will find and therefore we will be more confident about the quality of the software. Ideally we would be tempted to exhaustively test the input space. But as stated above, exhaustively testing the combinations of valid inputs will be impossible for most of the programs, let alone considering invalid inputs, timing, sequence, and resource variables. Combinatorial explosion is the major roadblock in functional testing. To make things worse, we can never be sure whether the specification is either correct or complete. Due to limitations of the language used in the specifications (usually natural language), ambiguity is often inevitable. Even if we use some type of formal or restricted language, we may still fail to write down all the possible cases in the specification. Sometimes, the specification itself becomes an intractable problem: it is not possible to specify precisely every situation that can be encountered using limited words. And people can seldom specify clearly what they want — they usually can tell whether a prototype is, or is not, what they want after they have been finished. Specification problems contributes approximately 30 percent of all bugs in software.

The research in black-box testing mainly focuses on how to maximize the effectiveness of testing with minimum cost, usually the number of test cases. It is not possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Partitioning is one of the common techniques. If we have partitioned the input space and assume all the input values in a partition is equivalent, then we only need to test one representative value in each partition to sufficiently cover the whole input space. Domain testing partitions the input domain into regions, and consider the input values in each domain an equivalent class. Domains can be exhaustively tested and covered by selecting a representative value(s) in each domain. Boundary values are of special interest. Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary value analysis requires one or more boundary values selected as representative test cases. The difficulties with domain testing are that incorrect domain definitions in the specification can not be efficiently discovered.

Notes Good partitioning requires knowledge of the software structure. A good testing plan will not only contain black-box testing, but also white-box approaches, and combinations of the two.

12.2.1 Black-box Testing Techniques

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. 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 attempts to find errors in the following categories:

1. incorrect or missing functions,
2. interface errors,
3. errors in data structures or external database access,

Notes

4. behaviour or performance errors, and
5. initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing.

Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

There are various techniques in developing test cases for black box testing. I will be explaining two of the most common techniques, Equivalence Partitioning and BVA (Boundary Value Analysis).

Equivalence Partitioning

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

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is boolean, one valid and one invalid class are defined.

Task Try to analyze the limitations of Equivalence Partitioning over BVA.

As an example, consider data maintained as part of an automated banking application.

The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form:

- area code—blank or three-digit number
- prefix—three-digit number not beginning with 0 or 1

Notes

- suffix—four-digit number
- password—six digit alphanumeric string
- commands—check, deposit, bill pay, and the like

The input conditions associated with each data element for the banking application can be specified as area code:

- Input condition, Boolean—the area code may or may not be present.
- Input condition, range—values defined between 200 and 999, with specific exceptions.
- ***prefix:*** Input condition, range—specified value >200
- Input condition, value—four-digit length
- ***password:*** Input condition, Boolean—a password may or may not be present.
- Input condition, value—six-character string.
- ***command:*** Input condition, set—containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

Designing Test Cases using Equivalence Partitioning

To use equivalence partitioning, you will need to perform two steps

1. Identify the equivalence classes
2. Design test cases

Step 1: Identify Equivalence Classes

Take each input condition described in the specification and derive at least two equivalence classes for it. One class represents the set of cases which satisfy the condition (the valid class) and one represents cases which do not (the invalid class)

Following are some general guidelines for identifying equivalence classes:

1. If the requirements state that a numeric value is input to the system and must be within a range of values, identify one valid class inputs which are within the valid range and two invalid equivalence classes inputs which are too low and inputs which are too high. For example, if an item in inventory can have a quantity of - 9999 to + 9999, identify the following classes:
   (a) *one valid class:* (QTY is greater than or equal to -9999 and is less than or equal to 9999). This is written as (- 9999 < = QTY < = 9999)
   (b) the invalid class (QTY is less than -9999), also written as (QTY < -9999)
   (c) the invalid class (QTY is greater than 9999), also written as (QTY >9999)
2. If the requirements state that the number of items input by the system at some point must lie within a certain range, specify one valid class where the number of inputs is within the valid range, one invalid class where there are too few inputs and one invalid class where there are, too many inputs.

Notes

Example: Specifications state that a maximum of 4 purchase orders can be registered against anyone product. The equivalence classes are: the valid equivalence class: number of purchase orders is greater than or equal to 1 and less than or equal to 4 , also written as (1 < = no. of purchase orders < = 4) the invalid class (no. of purchase orders> 4) the invalid class (no. of purchase orders < 1)

3. If the requirements state that a particular input item match one of a set of values and each case will be dealt with the same way, identify a valid class for values in the set and one invalid class representing values outside of the set. For example, if the requirements state that a valid province code is ON, QU, and NB, then identify : the valid class code is one of ON, QU, NB the invalid class code is not one of ON, QU, NB.

4. If the requirements state that a particular input item match one of a set of values and each case will be dealt with differently, identify a valid equivalence class for each element and only one invalid class for values outside the set. For example, if a discount code must be input as P for preferred customer, R for standard reduced rate, or N for none, and if each case is treated differently, identify

   the valid class code = P

   the valid class code = R

   the valid class code = N

   the invalid class code is not one of P, R, N

5. If you think any elements of an equivalence class will be handled differently than the others, divide the equivalence class to create an equivalence class with only these elements and an equivalence class with none of these elements. For example, a bank account balance may be from $0 up to $ 1,000,000, and balances $ 1,000 or over are not subject to service charges. Identify:

   the valid class: ($ 0 < = balance < $ 1,000) i.e., balance is between 0 and $ 1,000 - not including $ 1,000

   the valid class: ($ 1, 000 < = balance < = $ 1,000,000 i.e., balance is between $ 1,000 and $1,000,000 inclusive

   the invalid class: (balance < $ 0)

   the invalid class: (balance> $ 1,000,000)

A definition of Equivalence Partitioning from our software testing dictionary:

Equivalence Partitioning: An approach where classes of inputs are categorized for product or function validation. This usually does not include combinations of input, but rather a single state value based by class. For example, with a given function there may be several classes of input that may be used for positive testing. If function expects an integer and receives an integer as input, this would be considered as positive test assertion. On the other hand, if a character or any other input class other than integer is provided, this would be considered a negative test assertion or condition.

Boundary Value Analysis

BVA extends equivalence partitioning by focusing on data at the "edges" of an equivalence class.

Notes

For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center". It is for this reason that boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

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

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

It's widely recognized that input values at the extreme ends of input domain cause more errors in system. More application errors occur at the boundaries of input domain. 'Boundary value analysis' testing technique is used to identify errors at boundaries rather than finding those exist in center of input domain.

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

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

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

Boundary value analysis is often called as a part of stress and negative testing.

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

Notes

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

This should be a very basic and simple example to understand the Boundary value analysis and Equivalence partitioning concept.

Advantages of Black-box Testing

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

Disadvantages of Black-box Testing

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

12.3 Fault-based Testing

The strategy for fault based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis. The objective of fault based testing within an object oriented system is to design tests that have a higher possibility of uncovering plausible errors. Test cases are designed to exercise the design or code and to determine whether faults exist. This approach is really no better than any random testing technique if faults in object oriented systems are implausible.

Three types of faults are encountered when integration testing looks for plausible faults in operation calls or message connections. These faults are:

1. unexpected result.
2. wrong operation/message used.
3. incorrect invocation.

Integration testing applies to attributes as well as operations. Integration testing attempts to find errors in the client side and not the server.

1. Testing that employs a test data selection strategy designed to generate test data capable of demonstrating the absence of a set of pre-specified faults, typically, frequently occurring faults.
2. This type of testing allows for designing test cases based on the client specification or the code or both. It tries to identify plausible faults (areas of design or code that may lead to errors). For each of these faults a test case is developed to “flush” the errors out. These tests also force each line of code to be executed

Notes

Self Assessment

Fill in the blanks:

8. Black-box testing, also called ……. testing, focuses on the functional requirements of the software.
9. At times we might want to check the behavior when the value of one variable exceeds its maximum. This is called …… testing.
10. Black box testing is a software testing techniques in which …… of the software
11. An …… class represents a set of valid or invalid states for input conditions.
12. An approach where classes of …… are categorized for product or function validation.
13. Boundary value analysis leads to a selection of …….. that exercise bounding values.
14. Boundary value analysis is a test case design technique that complements ……..
15. Test cases are designed to exercise the design or code and to determine whether …… exist.

Caselet

BIM Course on Software Testing

THINKSOFT Global Services, a Chennai-based independent software testing provider, has signed an agreement with the Bharathidasan Institute of Management (BIM), Tiruchi, to offer a course on application software testing — a process used to identify the correctness, completeness and quality of developed computer software.

The course, which will start in July, will be an elective subject for second year students, according to the institute's Director (in-charge), Mr M. Sankaran.

This agreement is part of the institute's efforts to focus on industry-academia interaction. Mr Sankaran said such agreements would help bridge the gap between supply and demand for quality students in the industry. The course will help BIM students build domain expertise and management strength in software testing he told newspersons.

According to Ms Vanaja Arvind, Executive Director, Thinksoft, the company will provide the course curriculum as per BIM specifications.

12.4 Summary

- Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.
- White-box testing or glass-box testing is a test case design method that uses the control structure of the procedural design to obtain test cases.
- Using this methodology, a software engineer can come up with test cases that guarantee that all independent paths within a module have been exercised at least once; exercise all the logical decisions based on their true and false sides; executing all loops at their boundaries and within them; and exercise internal data structures to ensure their validity.

Notes

- Black box testing is a software testing techniques in which functionality of the Software Under Test (SUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software.
- The strategy for fault based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis.
- The objective of fault based testing within an object oriented system is to design tests that have a higher possibility of uncovering plausible errors.

12.5 Keywords

BVA: Boundary Value Analysis

SUT: Software Under Test

12.6 Review Questions

1. Why testing is more than just debugging? Explain with examples.
2. Explain why the reason of conducting white-box testing largely depends upon the nature of defects in software?
3. The test is accurate only if the tester knows what the program is supposed to do. Comment.
4. BVA extends equivalence partitioning by focusing on data at the "edges" of an equivalence class. Explain briefly.
5. Boundary value test cases are obtained by keeping one variable at its extreme and the other at its nominal value. Discuss with example.
6. How the equivalence classes can be defined?
7. While White Box Testing (Unit Testing) validates internal structure and working of your software code, the main focus of black box testing is on the validation of your functional requirements. Which of the two testing is more beneficial explain?
8. In case you push code fixes in your live software system, a complete system check (black box regression tests) becomes essential. Discuss.
9. Describe how white box testing has its own merits and help detect many internal errors which may degrade system performance?
10. In Black Box Testing we just focus on inputs and output of the software system without bothering about internal knowledge of the software program. Analyze.

Answers: Self Assessment

1.	program	2.	statements
3.	integration	4.	Cyclomatic
5.	inversely	6.	implementation
7.	system levels	8.	behavioral
9.	robustness	10.	functionality
11.	equivalence	12.	inputs

Notes

13. test cases
14. equivalence partitioning
15. faults

12.7 Further Readings

Books

Rajib Mall, *Fundamentals of Software Engineering*, Second Edition, PHI.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

R.S. Pressman, *Software Engineering a Practioner's Approach*, 5th edition, Tata McGraw Hill Higher education.

Sommerville, *Software Engineering*, Sixth edition, Pearson Education.

Online links

http://en.wikipedia.org/wiki/White-box_testing

http://en.wikipedia.org/wiki/Black-box_testing

Notes

Unit 13: Building the Analysis Model

CONTENTS

Objectives
Introduction
13.1 Building the Analysis Model
13.1.1 Data Modeling: ER Diagram
13.1.2 Relationship
13.1.3 Cardinality and Multiplicity
13.2 Requirement Analysis
13.2.1 Requirement Analysis Objective
13.2.2 Philosophy
13.3 Rules-of-Thumb Analysis
13.4 Domain Analysis
13.5 Summary
13.6 Keywords
13.7 Review Questions
13.8 Further Reading

Objectives

After studying this unit, you will be able to:

- Build a Analysis Model
- Scan Requirement Engineering
- Discuss Rules of Thumb Analysis
- Explain Domain Rule

Introduction

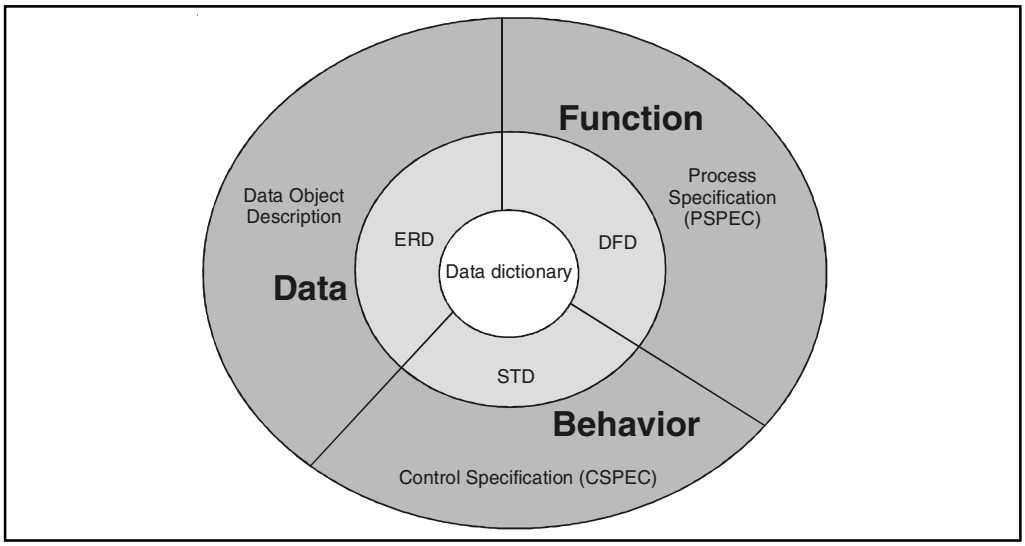
The analysis model explains the structure of the system or application that you are modeling. It consists of class diagrams and sequence diagrams that illustrate the logical implementation of the functional requirements that you identified in the use case model.

The analysis model recognizes the main classes in the system and contains a set of use case realizations that explain how the system will be built. Class diagrams explain the static structure of the system by using stereotypes to model the functional parts of the system. Sequence diagrams understand the use cases by explaining the flow of events in the use cases when they are executed. These use case realizations model how the parts of the system interact within the context of a specific use case.

Notes You can think of the analysis model as the foundation of the design model since it explains the logical structure of the system, but not how it will be implemented.

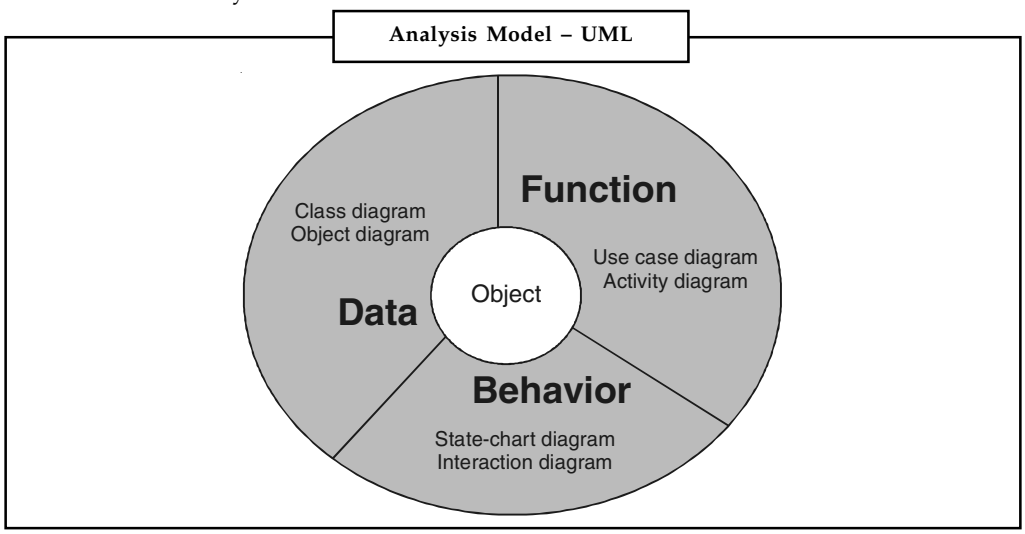
Notes

13.1 Building the Analysis Model



Did u know? **Why do we model?**

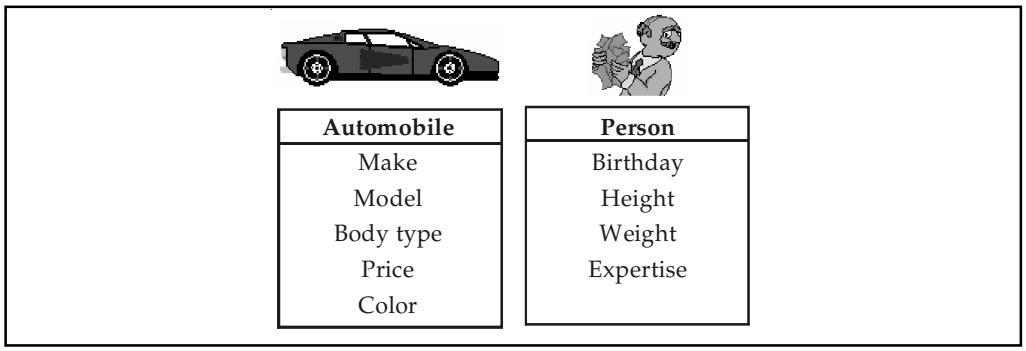
- We build models so that we can better understand the system we are developing.
- We build models of complex systems because we cannot comprehend such a system in its entirety.



13.1.1 Data Modeling: ER Diagram

1. Examines data objects **independently** of processing
2. Focuses attention on the **data domain**
3. Creates a model at the **customer**'s level of **abstraction**
4. Indicates how data objects **relate** to one another.

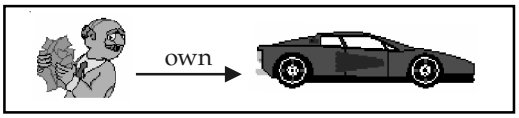
Notes



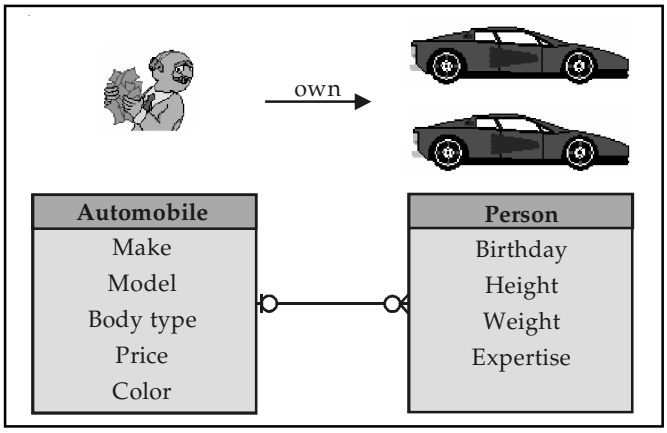
Task Explain what are the aims which are achieved by modeling?

13.1.2 Relationship

1. Connectedness
2. A fact that must be remembered by the system and cannot or is not computed or derived
   (a) Several instance of a relationship can exist
   (b) Entity can be related in many ways

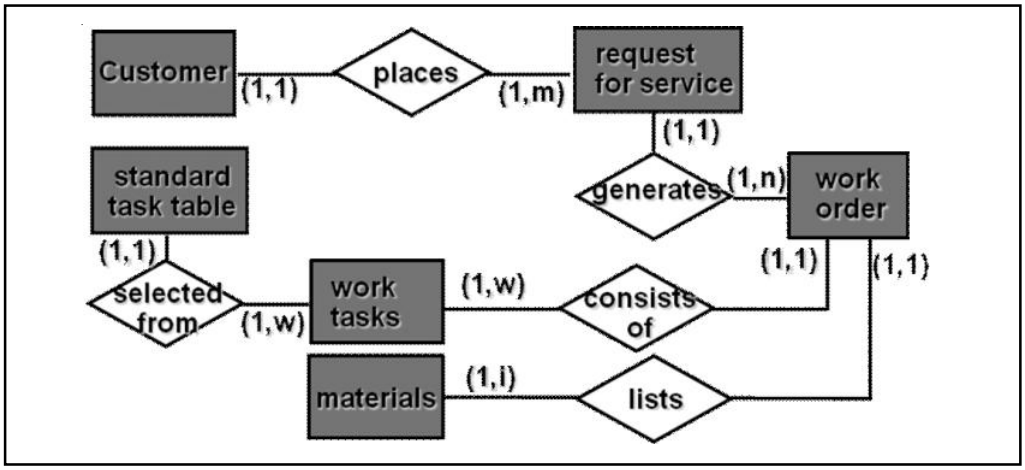


13.1.3 Cardinality and Multiplicity



Notes

An Example



Self Assessment

Fill in the blanks:

1. …… classes **inherit** their attributes and services from one or more super-classes. These may then be specialised as necessary
2. Data must always be process in some way to achieve …….

13.2 Requirement Analysis

Requirements analysis, also called requirements engineering, is the process of determining user expectations for a new or modified product. These features, called requirements, must be quantifiable, relevant and detailed. In software engineering, such requirements are often called functional specifications. Requirements analysis is an important aspect of project management.

Requirements analysis involves frequent communication with system users to determine specific feature expectations, resolution of conflict or ambiguity in requirements as demanded by the various users or groups of users, avoidance of feature creep and documentation of all aspects of the project development process from start to finish. Energy should be directed towards ensuring that the final system or product conforms to client needs rather than attempting to mold user expectations to fit the requirements.

Requirements analysis is a team attempt that demands a combination of hardware, software and human factors engineering expertise as well as skills in dealing with people.

In systems engineering and software engineering, requirements analysis encompasses those tasks that go into determining the requirements of a new or altered system, taking account of the possibly conflicting requirements of the various stakeholders, such as users. Requirements analysis is critical to the success of a project.

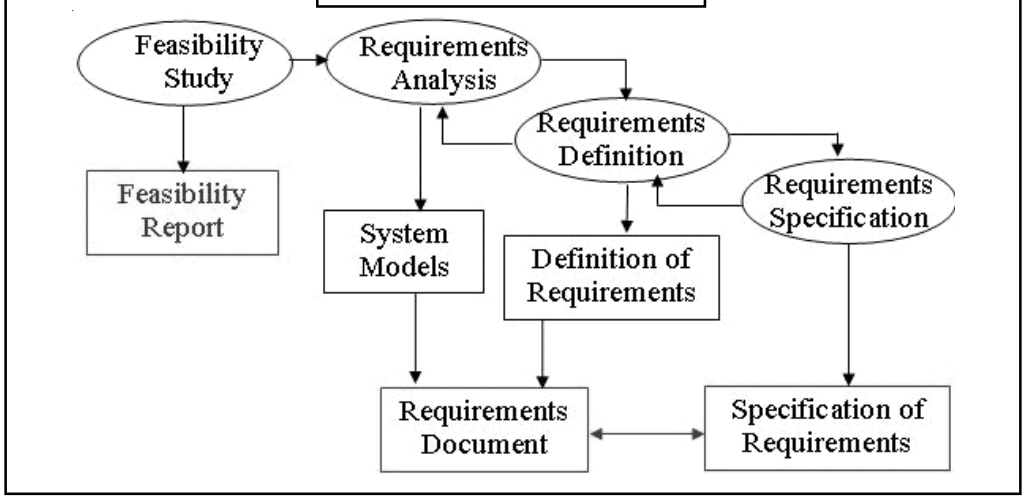
Systematic requirements analysis is also known as requirements engineering. It is sometimes referred to loosely by names such as requirements gathering, requirements capture, or requirements specification. The term "requirements analysis" can also be applied particularly to the analysis proper (as opposed to elicitation or documentation of the requirements, for instance).

Notes

Did u know? **What are the basics of requirements?**

Requirements must be measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

Figure 13.1: Requirements Process



13.2.1 Requirement Analysis Objective

The objective of the requirements analysis phase is to recognize and secure approval for a project requirements specification.

A complete requirements analysis is needed to ensure fast project delivery and optimal return on information management investment.

13.2.2 Philosophy

This topic is concerned with the process of analyzing requirements to

- Detect and resolve conflicts between requirements
- Discover the bounds of the software and how it must interact with its environment
- Elaborate system requirements to derive software requirements

The conventional view of requirements analysis has been that it be reduced to conceptual modeling using one of a number of analysis methods such as the Structured Analysis and Design Technique (SADT). While conceptual modeling is significant, we include the classification of requirements to help inform trade-offs between requirements (requirements classification) and the process of establishing these trade-offs (requirements negotiation).

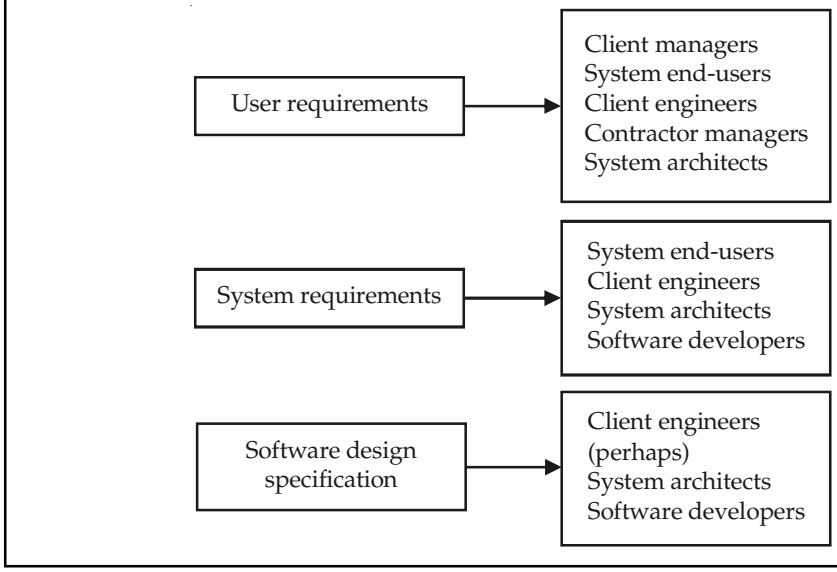
Caution Care must be taken to explain requirements precisely enough to enable the requirements to be validated, their implementation to be verified, and their costs to be estimated.

Notes

Requirements Classification

Requirements can be classified on a number of dimensions:

Figure 13.2: Requirement types



Other classifications may be appropriate, depending upon the organization's usual practice and the application itself.

There is a sturdy overlap between requirements classification and requirements attributes.

Conceptual Modeling

The growth of models of a real-world problem is key to software requirements analysis. Their purpose is to aid in understanding the problem, rather than to initiate design of the solution. Hence, conceptual models comprise models of entities from the problem domain configured to reflect their real-world relationships and dependencies. Several kinds of models can be developed. These include data and control flows, state models, event traces, user interactions, object models, data models, and many others. The factors that influence the choice of model include

- The nature of the problem. Some types of software demand that certain aspects be analyzed particularly rigorously.

Example: Control flow and state models are likely to be more important for real-time software than for management information software, while it would usually be the opposite for data models.

- The expertise of the software engineer. It is often more productive to adopt a modeling notation or method with which the software engineer has experience.
- The process requirements of the customer. Customers may impose their favored notation or method, or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.
- The availability of methods and tools. Notations or methods which are poorly supported by training and tools may not achieve widespread acceptance even if they are suited to particular types of problems.

Notes

Notes In almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment. This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with the environment.

The matter of modeling is tightly coupled with that of methods. For practical purposes, a method is a notation (or set of notations) supported by a process which guides the application of the notations. There is little empirical proof to support claims for the superiority of one notation over another. However, the widespread acceptance of a exacting method or notation can lead to beneficial industry-wide pooling of skills and knowledge. This is currently the situation with the UML (Unified Modeling Language). Formal modeling using notations based on discrete mathematics, and which are traceable to logical reasoning, have made an impact in some specialized domains. These may be imposed by customers or standards or may offer compelling advantages to the analysis of certain critical functions or components.

Architectural Design and Requirements Allocation

Sometime, the architecture of the solution must be derived. Architectural design is the point at which the requirements process overlaps with software or systems design and illustrates how impossible it is to cleanly decouple the two tasks. This topic is intimately related to the Software Structure and Architecture subarea in the Software Design KA. In many cases, the software engineer acts as software architect because the process of analyzing and elaborating the requirements demands that the components that will be responsible for satisfying the requirements be identified. This is requirements allocation–the assignment, to components, of the responsibility for satisfying requirements.

Allocation is significant to permit detailed analysis of requirements. Hence, for example, once a set of requirements has been allocated to a component, the individual requirements can be further analyzed to discover further requirements on how the component needs to interact with other components in order to satisfy the allocated requirements. In large projects, allocation stimulates a new round of analysis for each subsystem.

Architectural design is closely recognized with conceptual modeling. The mapping from real-world domain entities to software components is not always obvious, so architectural design is identified as a separate topic.

Task Analyze how the requirements of notations and methods are broadly the same for both conceptual modeling and architectural design.

Requirements Negotiation

Another term usually used for this sub-topic is "conflict resolution." This concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and non-functional requirements. In most cases, it is foolish for the software engineer to make a unilateral decision, and so it becomes necessary to consult with the stakeholder(s) to reach a consensus on an appropriate trade-off. It is often important for contractual reasons that such decisions be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for considering it a requirements validation topic.

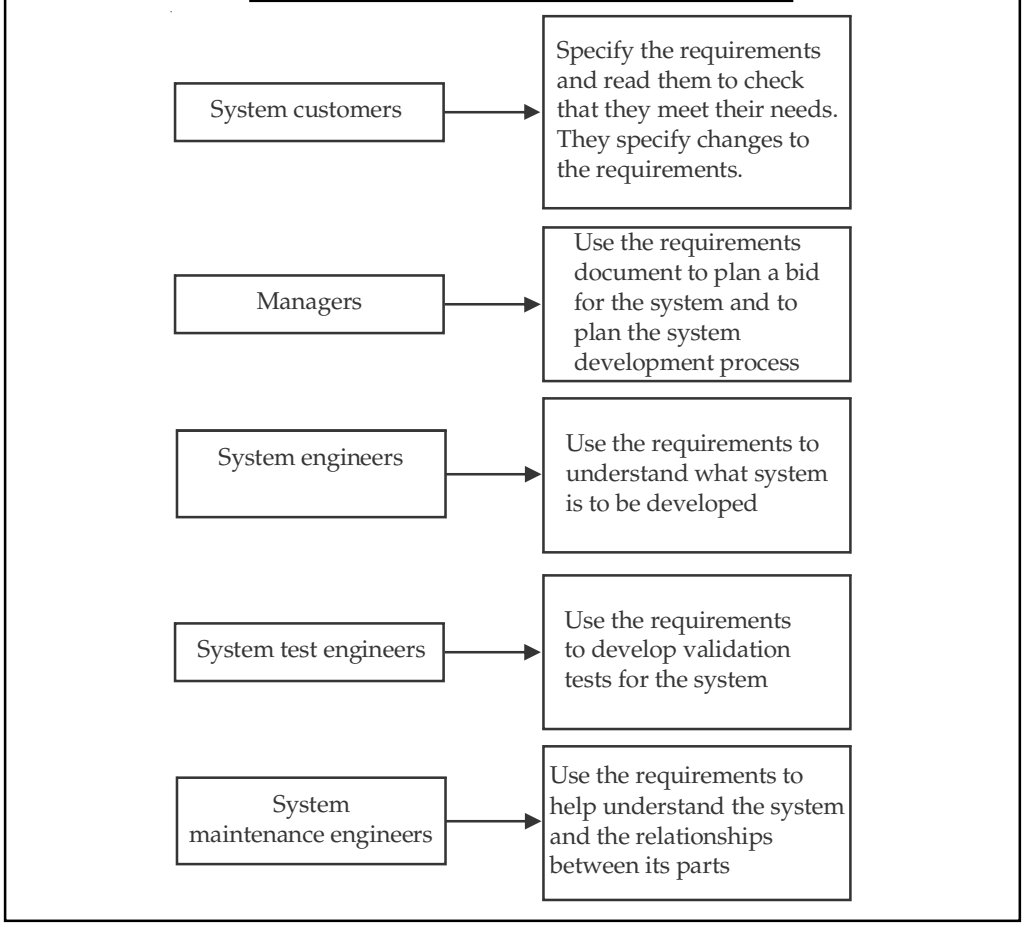
Notes

Requirements Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. Typical physical systems have a relatively small number of such values. Typical software has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements. So, in software engineering jargon, "software requirements specification" typically refers to the production of a document, or its electronic equivalent, which can be systematically reviewed, evaluated, and approved.

For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required.

Figure 13.3: Types of Requirement Document



There are some approaches to requirements specification:

- Natural language
- Structured natural language
- Design description language
- Requirements specification language

Notes

- Graphical notation
- Formal specification

The System Definition Document

This document (occasionally known as the user requirements document or concept of operations) records the system requirements. It describes the high-level system requirements from the domain perspective. Its readership includes representatives of the system users/customers (marketing may play these roles for market-driven software), so its content must be understood in terms of the domain. The document lists the system requirements along with background information about the overall objectives for the system, its target environment and a statement of the constraints, assumptions, and non-functional requirements.

Notes It may include conceptual models designed to illustrate the system context, usage scenarios and the principal domain entities, as well as data, information, and workflows.

System Requirements Specification

Developers of systems with substantial software and non-software components, a modern airliner, for example, often separate the description of system requirements from the description of software requirements. In this view, system requirements are specified, the software requirements are derived from the system requirements, and then the requirements for the software components are specified. Strictly speaking, system requirements specification is a systems engineering activity and falls outside the scope of this Guide.

Software Requirements Specification

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do, as well as what it is not expected to do. For non-technical readers, the software requirements specification document is often accompanied by a software requirements definition document.

Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules.

Organizations can also use a software requirements specification document to develop their own validation and verification plans more productively.

Software requirements specification provides an informed basis for transferring a software product to new users or new machines. Finally, it can provide a basis for software enhancement.

Software requirements are often written in natural language, but, in software requirements specification, this may be supplemented by formal or semi-formal descriptions. Selection of appropriate notations permits particular requirements and aspects of the software architecture to be explained more precisely and concisely than natural language. The general rule is that notations should be used which allow the requirements to be explained as precisely as possible. This is particularly crucial for safety-critical and certain other types of dependable software. However, the choice of notation is often constrained by the training, skills and preferences of the document's authors and readers.

Notes

A number of quality indicators have been developed which can be used to relate the quality of software requirements specification to other project variables such as cost, acceptance, performance, schedule, reproducibility, etc. Quality indicators for individual software requirements specification statements include imperatives, directives, weak phrases, options, and continuances. Indicators for the entire software requirements specification document include size, readability, specification, depth, and text structure.

Self Assessment

Fill in the blanks:

3. Energy should be directed towards ensuring that the final system or product conforms to client needs rather than attempting to mold user expectations to fit the……..
4. Requirements analysis is a …… that demands a combination of hardware, software and human factors engineering expertise as well as skills in dealing with people.
5. A …….. requirements analysis is needed to ensure rapid project delivery and optimal return on information management investment.
6. …….. or methods which are poorly supported by training and tools may not achieve widespread acceptance even if they are suited to particular types of problems.
7. The document lists the system requirements along with …… information.
8. Organizations can also use a software requirements specification document to develop their own ……and…….. plans more productively.
9. …….. has a large number of requirements, and the emphasis is shared.

13.3 Rules-of-Thumb Analysis

Use the Rules-of-Thumb Analysis window to examine specific trace data in the Performance Warehouse data tables of the performance database using the selected rule of thumb or all rules of thumb in the selected cluster, and to view the analysis result.

1. The model should focus on requirements that are visible within the problem or business domain.
   - The level of abstraction should be relatively high.
2. Each element of the analysis model should
   - add to an overall understanding of software requirements
   - provide insight into the
   - information domain
   - function of the system
   - behavior of the system
3. Delay consideration of infrastructure and other non-functional models until design.
4. Minimize coupling throughout the system.
5. Be certain that the analysis model provides value to all stakeholders.
6. Keep the model as simple as it can be.

Notes

Task Visit a website related to rules of thumb analysis and analyze the limitations in a group of four.

Self Assessment

Fill in the blank:

10. The model should focus on requirements that are visible within the problem or ……...

13.4 Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, usually for reuse on multiple projects within that application domain [Object oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

- Define the domain to be investigated.
- Collect a representative sample of applications in the domain.
- Analyze each application in the sample.
- Develop an analysis model for the objects.

Target-specific OO analysis techniques often generate models with applicability stretching beyond the needs of the system(s) under consideration, and thus intrinsically incorporate at least some form and extent of domain analysis. However, the notion of domain analysis as a distinguishable enterprise remains an immature topic, in need of considerable development. In this unit, we survey general views, models, and variants of domain analysis, along with their consequences for reuse.

There are several ways to define "domain". For example,

- A collection of current and future (software) applications that share a set of common characteristics.
- A well-defined set of characteristics that accurately, narrowly, and completely explain a family of problems for which computer application solutions are being, and will be sought.

A founder, if not the founder, of domain analysis is Neighbors. He wrote in 1980:

The key to reusable software is captured in domain analysis in that it stresses the reusability of analysis and design, not code.

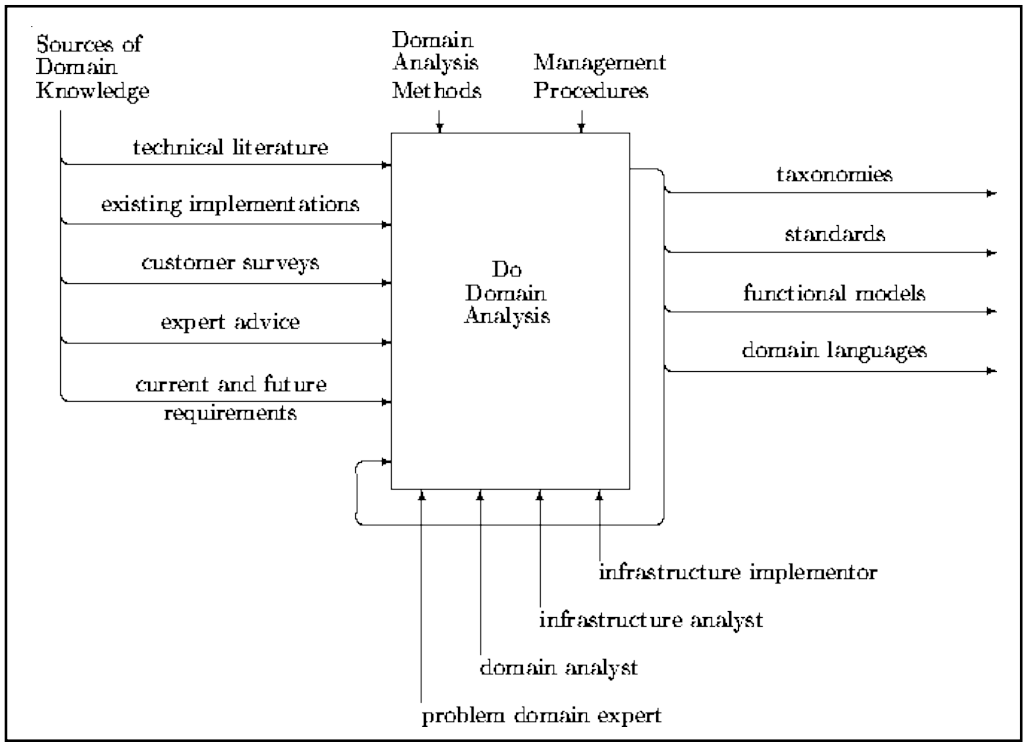
Models

The scope of a domain investigation can vary extensively. A definition of domain analysis formulated by Prieto-Diaz elucidates its purpose as:

A process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems.

Notes

Arango and Prieto-Diaz present a model of domain analysis summarized in the following SADT diagram:



This model explains domain analysis as an activity that takes multiple sources of input, produces many different kinds of output, and is heavily parameterized. For example, one parameter is the development paradigm (e.g., SA, Jackson, OO). Raw domain knowledge from any relevant source is taken as input. Participants in the process can be, among others, domain experts and analysts. Outputs are (semi) formalized concepts, domain processes, standards, logical architectures, etc. Subsequent activities produce generic design fragments, frameworks, etc.

Notes While this account gives an inspiring initial insight into domain analysis, it is not the full story. Several refinements are presented next.

Product Definition Domain Analysis

When a product is seen as part of a new or an existing stream of products, the domain of this product stream may itself be studied. This study will in general go beyond technical aspects of the product.

Example: Strategic alignment, longer term marketing aspects, product positioning, risk analysis, common look-and-feel, covering a diversity of product features, etc., will play a role in conjunction with investigations of generic terminology, logical architectures, reliability standards, and other general considerations.

Notes

Requirements Domain Analysis

When there is sufficient confidence that a stream of products can be produced, one may want to factor out the commonalities in the multiple analyses that must be done for each product. Thus one may want to do a conceptual domain analysis that yields ordinary ground for each specific analysis. OO analysis notions lend themselves for capturing generic concepts at multiple levels of granularity. Ensembles, subensembles, classes, and generic relationships are all candidates for describing an application domain.

While we can use the notions and notations from an OO analysis method for requirements domain analysis, we have to adjust the process dimension. We cannot rely on a system-specific requirements document as input to the process. Instead, we have to take in any relevant features from the documentation that explains the commonality of the products. Experts and customers may be tapped, as suggested in the generic diagram. Though, the situation differs from the diagram in that people have to be primed for more specific and detailed information. The output side differs as well because the process stops earlier; no model is to be constructed. Instead, generic classes, relationships, ensembles, etc., are produced. These may be organized into one or more OO frameworks that may be specialized to the needs of particular systems.

Example: Many of the ATM examples in previous units are not geared to any specific system. To the extent to which these descriptions are realistic, they are contributions to an OO domain analysis of "ATMs for banks".

Our model of the OOA process in unit represents an even better instance of this form of domain analysis. This model abstracted across different development styles and contexts. It did not culminate in a particular target model, but only those model components forming a basis for any OOA process.

Domain Engineering

A requirements domain analysis may lead to an OO domain engineering effort. This entails the construction of design fragments of the generic elements identified by a requirements domain analysis. These designs can be implemented and added to a domain-specific code library.

Generator Domain Analysis

When a stable domain serves as the basis of a product line or market segment, one may consider constructing a generator for a particular domain. This generator may then be used to automatically build (parts of) any of a series of related products. Relational database systems are an example of a mature, stable domain where it is quite conceivable to perform a generator type domain analysis. The query language, platform, operating system and windowing environment would be main parameters for such a relational database builder.

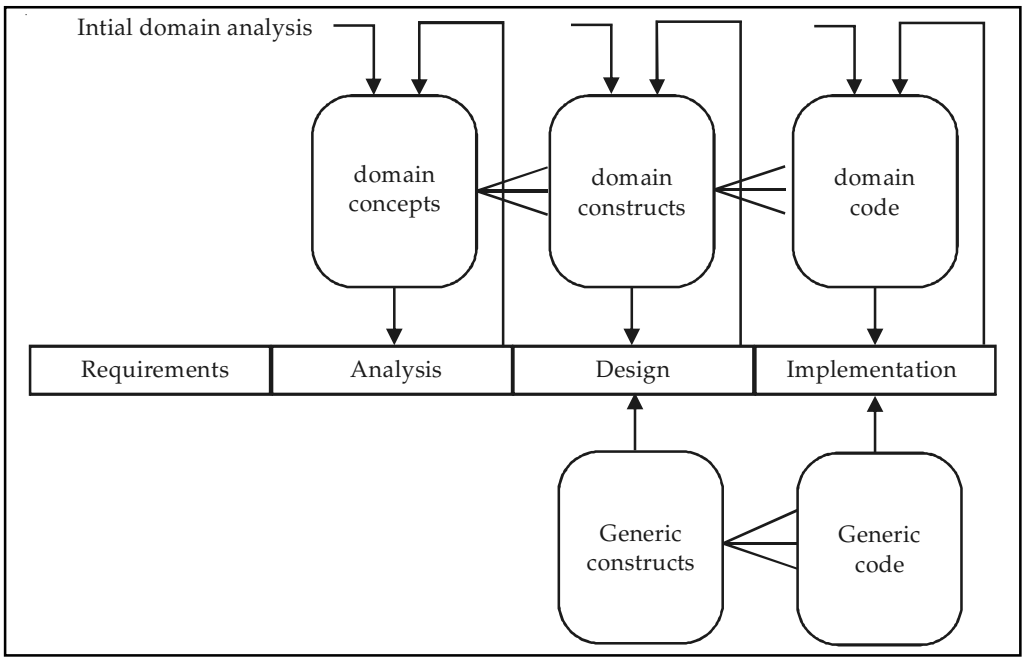
The analysis performed for the construction of such a meta-program may be seen as a third version of the notion of domain analysis. One may assume for such an enterprise not only that the domain is stable and well understood, but also that domain specific design and/or code libraries are available. One may even step one level higher. '

Example: The Rose system (Reuse of Software Elements) was an experimental meta-meta-program that assisted in capturing domain knowledge and design know-how for the domain.

Notes

Reuse

Domain analysis is not a one-shot affair. Product definitions evolve continuously. The development of a particular system that exploits previously accumulated domain knowledge can be the source for new insights about the domain that adds to or refines codified domain knowledge. In analogy to the emergence of domain-specific code libraries, we foresee the development of domain-specific analysis concept repositories, linked ultimately to code via domain-specific design repositories. The following diagram explains the interactions:



Functional model components are the primary outputs of a domain analysis. The feedback loops explain/prescribe that the outputs of the different phases are to be abstracted and added to the domain repositories. The "bird-feet" lines in the diagram that are attached to the repositories express their interconnections. For example, a domain analysis concept can have multiple realizations in the design repository. Similarly, a domain construct can have multiple realizations in the corresponding code repository, where each realization satisfies different auxiliary requirement trade-offs.

Domain analysis is the spearhead for disciplined reuse in software development. This is quite obvious for the generator version of domain analysis, but applies as well to the two weaker versions. An organization for system development will be complemented, when cost effective, by an organization that maintains and manages domain-specific repositories. OO analysis has much to offer to domain analysis from a technical perspective. However the sociological, cultural and organizational problems of realizing a cost effective reuse program that underlies our diagram extend beyond the technical dimension. This has led to the widespread adoption of reuse-based strategies in OO design and programming efforts. However, these practices remain incomplete without equally prevalent adoption of a reuse-based engineering discipline at the requirements and analysis levels.

Self Assessment

Notes

Fill in the blanks:

11. …….. analysis techniques often generate models with applicability stretching beyond the needs of the system(s)
12. The key to …….. software is captured in domain analysis in that it stresses the reusability of analysis and design, not code.
13. ……. model components are the primary outputs of a domain analysis.
14. OO analysis has much to offer to ……. from a technical perspective.
15. The analysis performed for the construction of such a meta-program may be seen as a third version of the …….. of domain analysis.

Caselet

Cost Analysis Port to Port

BARREN rock. That is how Hong Kong was explaind over 150 years ago. It has no natural resources, except one of the finest deep-water ports in the world. Also, one of the busiest, because an estimated $150 billion goods flow in and out of HK.

In terms of total container throughput in 2002, HK scored 19.1 million TEUs (that is, 20-foot-equivalent unit), while it was only 6.5 millions for Rotterdam and 4.8 millions for Antwerp. It derives 20 per cent of its GDP and employment from trade sector.

Interesting facts, but the problem is that it is facing competition, from nowhere else but its own parent, mainland China, observes the latest issue of The McKinsey Quarterly, through a cost analysis between doing business through HK versus ports in mainland China.

A paper by T. C. Chu, Alan Lau, and Nicolas C. Leung notes that though HK has played a historical role in linking the manufacturers of greater China with the consumers of developed nations, its dominance is being eroded by ports of Yantian and Shekou-Chiwan in southern China's Pearl River Delta, across the border from Hong Kong. With lower costs and improved service, those ports are attracting shippers.

A neat study for cost watchers, because there are cost elements that are uniform, and the ones that vary.

An obvious candidate for HK to trim is trucking costs, "from a typical factory in Dongguan — the Pearl River Delta's largest export-oriented manufacturing area."

For HK it is $370, which is more than double of what it costs to truck to the other two ports. McKinsey points out that this disparity accounts for "almost two-thirds of HK's 8 to 9 per cent cost disadvantage."

Though there are non-addressable issues such as wage because regulations prevent mainland China's truck drivers from entering HK, the authors offer a few suggestions to tackle at least a chunk of the cost difference.

Vehicles headed to HK from Dongguan make but one round trip a day, while the ones going to Yantian or Shekou-Chiwan can make two or three. Reason: Long waiting times at Customs and heavy traffic, consuming almost a third of the total trip time.

Contd…

Notes

To overcome these, the paper offers a few suggestions. These include the establishment of 24-hour Customs operations for increasing capacity utilisation, and improving the efficiency of the barges used to transport some of the containers down the delta to Hong Kong.

These measures have a cost-saving potential of $90 per container. "If the Guangdong authorities were to forgo licensing revenues and cut cross-border registration fees, an additional $30 per container per trip could be saved."

Subtract from the $220 gap (between $370 and $150 of trucking costs), $90 and $30, still you would have a $100 hole.

Such a cost chasm can easily tilt one's decision in favour of southern China's ports, but there are other advantages where HK scores. Such as: Easier Customs clearance, fewer inspections, more consistent demands for documentation, and more destinations served more frequently (4 to 20 times more trips to other places in Asia and 3 times as many to Europe and the US).

"Moreover, Hong Kong's status as a duty-free port makes it attractive as a regional warehouse for high-value goods, while its sophisticated legal and financial systems facilitate trade," states the paper while reiterating, "Nevertheless, Hong Kong needs to reduce trucking costs to maintain its competitiveness."

A focus on reducing terminal charges or expanding the port's capacity by building new container terminals — something the government and Shippers' Council are occupied with — is not the key, according to McKinsey, because "terminal charges account for only 10 per cent of total logistics costs".

Commonsense message, therefore, is to make existing terminals more efficient rather than creating additional capacity. And get the goods to port by streamlining logistics.

A cost-analysis model that can be ported on to closer home.

13.5 Summary

- While OO analysis is focused on the features and functionality of a single system to be generated, a domain analysis focuses on the common and variant features across a family of systems.
- At least three versions of domain analysis may be distinguished: (1) at the level of product definitions, (2) at the level of analysis of the proto-products, and (3) at the level of the analysis for a generator of applications in the domain. Because of the informality of version (1), the OO paradigm plays a less significant role there than in versions (2) and (3).
- To ensure the reusability of the domain models produced, domain analysts use diverse sources of domain knowledge.
- These sources provide information on the range of potential systems in the domain.
- While all domain analysis methods involve extraction of terminology and identification of a common domain language, the OO domain analysis method accomplishes this task through identification of classes, relationships and behaviors.
- Such domain models may translate into reusable frameworks.
- In a reuse strategy the domain analysis work-products must be maintained and enhanced over many systems.
- The domain analysis repository contains domain models that form the basis of subsequent systems analysis activities.

Notes

13.6 Keywords

OO: Object Oriented

UML: Unified Modeling Language

SADT: Structured Analysis and Design Technique

13.7 Review Questions

1. What are the problems faced by the organization to build the analysis model?
2. Requirements analysis is an important aspect of project management. Explain with reasons.
3. Systematic requirements analysis is also known as requirements engineering. Comment.
4. The objective of the requirements analysis phase is to identify and secure approval for a project requirements specification. Discuss.
5. The development of models of a real-world problem is key to software requirements analysis. Explain with examples.
6. Why delay consideration of infrastructure and other non-functional models until design?
7. Perform a domain analysis in any of the interpretations of this units for a domain with which you are familiar.
8. Explain how software domain analysis is the identification, analysis, and specification of common requirement?
9. OO analysis notions lend themselves for capturing generic concepts at multiple levels of granularity. Comment.
10. The domain analysis repository contains domain models. Analyze.

Answers: Self Assessment

1.	Object	2.	system function
3.	requirements	4.	team effort
5.	complete	6.	Notations
7.	Background	8.	validation and verification
9.	Typical software	10.	business domain
11.	Target-specific OO	12.	Reusable
13.	Functional	14.	domain analysis
15.	notion		

Notes

13.8 Further Reading

Books

E. Berard. *Essays in Object-Oriented Software Engineering*. Prentice Hall, 1992.

G. Arango and R. Prieto-Diaz. *Domain analysis: Concepts and research directions*. In R. Prieto-Diaz and G. Arango, editors, Domain Analysis: Acquisition of Reusable Information for Software Construction. IEEE Computer Society Press, May 1989.

M.D. Lubars. *Wide-spectrum support for software reusability.* In Workshop on Software Reusability and Maintainability. National Institute of Software Quality and Productivity, October 1987.

M.D. Lubars. Domain analysis and domain engineering in idea. Technical Report STP-295-88, MCC, September 1988.

Online links

http://www.objs.com/x3h7/a&d.htm'

http://en.wikipedia.org/wiki/Requirements_analysis

Notes

Unit 14: Data Modeling

CONTENTS

Objectives

Introduction

14.1 Data Models

14.2 Data Objects

14.3 Attribute

14.4 Relationships and Relationship Set

14.5 Cardinality and Modality

14.6 Summary

14.7 Keywords

14.8 Review Questions

14.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Recognize the Data Modeling concepts
- Describe Data Objects
- Explain Data Attributes
- Discuss Relationships
- Scan Cardinality and Modality

Introduction

A data model is not just a way of structuring data: it also defines a set of operations that can be performed on the data. The relational model, for example, defines operations such as select, project, and join. Although these operations may not be explicit in a particular query language, they provide the foundation on which a query language is built.

Example: Databases include word processors, dictionary, mail merges, user preferences and information management systems.

Various techniques are used to model data structure. Most database systems are built around one particular data model, although it is increasingly common for products to offer support for more than one model. For any one logical model various physical implementations may be possible, and most products will offer the user some level of control in tuning the physical implementation, since the choices that are made have a significant effect on performance. An example of this is the relational model: all serious implementations of the relational model allow the creation of indexes which provide fast access to rows in a table if the values of certain columns are known.

Notes

14.1 Data Models

To design any system, we require a mathematical model. Similarly, for DB Systems there are several models being used as given below:

- Hierarchical Model
- Network Model
- Relational Model
- Object-relational Model
- Object Model

Out of all the above models, the Relational Model is used by most of the database management packages, including Oracle, Ingres, Sybase, etc.

Hierarchical, Network, and Relational Models

Data pertaining to the real world applications are not isolated entities rather they are related to each other in many ways – simple and complex. A data model captures the essence of relation with the data items. Various data models view data and their relationships in different ways. In the following section we discuss the first two conventional data models – Hierarchical and Network – in reverse order.

Relational Model

The model was first introduced by Tod Codd of IBM Research in 1970. It uses the concept of a mathematical relation. Hence, the database is considered as a collection of relations. A relation can be thought as a table of values, each row in the table represents a collection of related data values. In relational model terminology, a row in the table is called a tuple and a column header is called an attribute.

A relation schema R, denoted by $R(A_1,...,A_n)$, is made up of a relation name R and a list of attributes $A_1,...,A_n$. The domain of A_i, denoted by $dom(A_i)$ is the set of all the possible values this attribute may take.

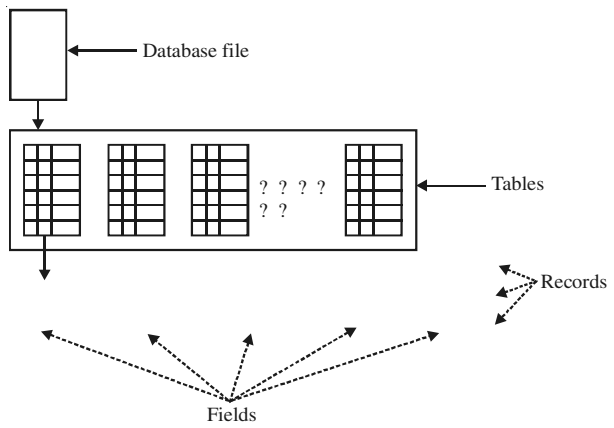
A relation schema that describes a relation R is called the name of this relation. The degree of a relation is the number of attributes in its relation schema.

Caution A record contains information regarding a single person or thing of interest.

A field stores data regarding a particular aspect of that person or thing. The organization is illustrated below:

Notes

Figure 14.1: Structure of a Relational Model



Network Data Model

The network model is a database model conceived as a flexible way of representing objects and their relationships. The network model original inventor was Charles Bachman, and it was developed into a standard specification published in 1969 by the CODASYL Consortium.

Where the hierarchical model structures data as a tree of records, with each record having one parent record and many children, the network model allows each record to have multiple parent and child records, forming a lattice structure.

The chief argument in favour of the network model, in comparison to the hierarchic model, was that it allowed a more natural modeling of relationships between entities. Although the model was widely implemented and used, it failed to become dominant for two main reasons. Firstly, IBM chose to stick to the hierarchical model with semi-network extensions in their established products such as IMS and DL/I. Secondly, it was eventually displaced by the relational model, which offered a higher-level, more declarative interface.

Notes Until the early 1980s the performance benefits of the low-level navigational interfaces offered by hierarchical and network databases were persuasive for many large-scale applications, but as hardware became faster, the extra productivity and flexibility of the relational model led to the gradual obsolescence of the network model in corporate enterprise usage.

Hierarchical Data Model

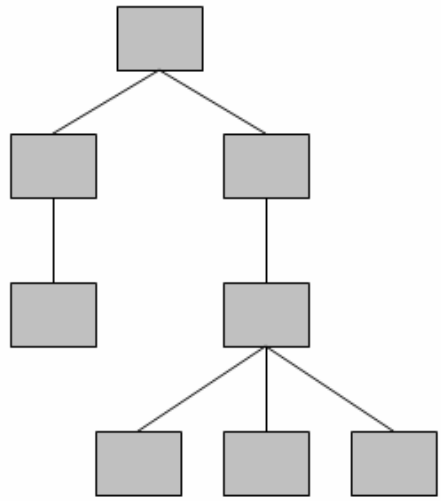
A hierarchical data model is a data model in which the data is organized into a tree-like structure. The structure allows repeating information using parent/child relationships: each parent can have many children but each child only has one parent. All attributes of a specific record are listed under an entity type. In a database, an entity type is the equivalent of a table; each individual record is represented as a row and an attribute as a column. Entity types are related to each other using 1: N mapping, also known as one-to-many relationships. The most recognized example of hierarchical model database is an IMS designed by IBM.

Hierarchical data model organizes data into a tree-like structure where each node has a single parent node, the root node being an exception in the sense that it does not have any parent.

Notes

While a node can have any number of child nodes, it can have only one parent node. A node in the tree represents a record. All attributes of a specific record are listed under an entity type. Entity types are related to each other using 1: N mapping. A typical hierarchical data model looks like the one shown in figure 14.2.

Figure 14.2: A Hierarchical Data Model



Prior to the development of the first Database Management System (DBMS), access to data was provided by application programs that accessed flat files. The data integrity problem and the inability of such file processing systems to represent logical data relationships lead to the first data model: the hierarchical data model. This model, which was implemented primarily by IBM's Information Management System (IMS) only allows one-to-one or one-to-many relationships between entities. Any entity at the many end of the relationship can be related only to one entity at the one end.

Example: A hierarchical data model would be if an organization had records of employees in a table (entity type) called "Employees". In the table there would be attributes/columns such as First Name, Last Name, Job Name and Wage. The company also has data about the employee's children in a separate table called "Children" with attributes such as First Name, Last Name, and date of birth. The Employee table represents a parent segment and the Children table represents a Child segment. These two segments form a hierarchy where an employee may have many children, but each child may only have one parent.

Self Assessment

Fill in the blanks:

1. A relation can be thought as a table of values, each row in the table represents a collection of related…….
2. A ……. is not just a way of structuring data: it also defines a set of operations that can be performed on the data.
3. A data model captures the essence of …….. with the data items.
4. In relational model terminology, a row in the table is called a …….. and a column header is called an attribute.

Notes

5. Although …….. was widely implemented and used, it failed to become dominant for two main reasons.
6. All attributes of a specific record are listed under an ……. type.
7. The degree of a relation is the number of ……. in its relation schema.

14.2 Data Objects

Now you know and understand the entities and relationships in your database, which is the most important part of the relational-database design process. After you determine the entities and relationships, a method that displays your thought process during database design might be helpful.

Most data-modeling methods provide some way to graphically display the entities and relationships.

Did u know? **Did IBM also use E-R Diagrams?**

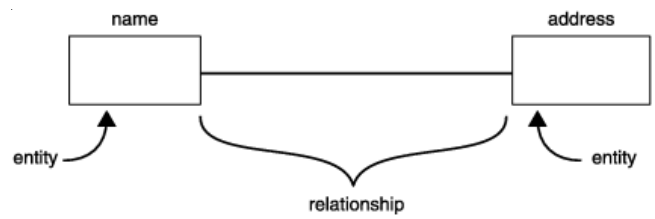
Yes, IBM Informix documentation uses the E-R diagram approach that C. R. Bachman originally developed.

E-R diagrams serve the following purposes. They:

- Model the informational requirements of an organization
- Identify entities and their relationships
- Provide a starting point for data definition (data-flow diagrams)
- Provide an excellent source of documentation for application developers and both database and system administrators
- Create a logical design of the database that can be translated into a physical schema

Several different styles of E-R diagrams exist. If you already have a style that you prefer, use it. Figure 14.3 shows a sample E-R diagram.

Figure 14.3. Symbols of an Entity-Relationship Diagram

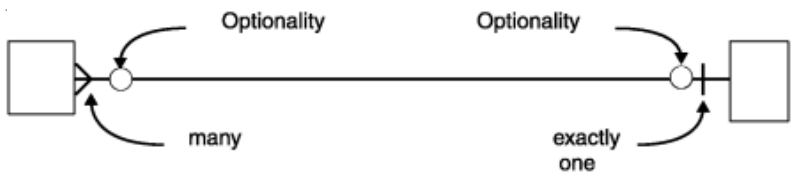


In an E-R diagram, a box represents an entity. A line represents the relationships that connect the entities. In addition, Figure 14.4 shows how you use graphical items to display the following features of relationships:

- A circle across a relationship link indicates *optionality* in the relationship (zero instances can occur).
- A small bar across a relationship link indicates that exactly one instance of the entity is associated with another entity (consider the bar to be a 1).
- The crow's-feet represent many in the relationship.

Notes

Figure 14.4 The parts of a Relationship in an Entity-Relationship Diagram



Self Assessment

Fill in the blanks:

8. Most data-modeling methods provide some way to …….. display the entities and relationships.
9. Create a logical design of the database that can be translated into a …….. schema.

14.3 Attribute

An attribute is a characteristic property of an entity. Each entity of the system under consideration has a number of different attributes highlighting different aspect of the entity. Thus, Name, Address, Date of birth, Date of opening account etc.; and Name, Address, Employee number, Date of joining etc.; are a few attributes of the Account holder and Branch manager entities respectively. Similarly, Account number, Branch code, etc., are a few attributes of the cheque book entity.

An entity is represented by the set of attributes it has. Every entity is described by a set of (attribute, data value) pair. Thus, the Account holder and Branch manager entities can be represented as follows:

AccountHolder(Name="Eknath", Address="A-45, Tagore Garden", DateOfBirth="13/11/1966", DateOfOpeningAccount= "12/10/2005")

BranchManager(Name="Vibhor Kasliwal",Address="B-46,Naraina",Employee Number= "BM008", DateOfJoining="01/07/2006")

An attribute may be restricted to take values from a set of values called domain of the attribute.

Attributes can be classified in a number of different ways. According to atomicity an attribute can be Simple or Composite. A simple attribute is atomic and cannot be broken further down into simpler components where as a composite attribute is composed of simpler values. For instance, while age is a simple attribute of a student entity, date of birth is a composite attribute because it can be still broken into three simpler values – day, month and year.

On the basis of whether an attribute must take a value for an instance of a given entity or not, attributes can be either Null-valued or Non-Null-valued. A null attribute may not take a value in which case it is said to have missing value. A non-null attribute must not have missing value. Note that missing value is not same as zero. Thus, the PhoneNumber attribute is a null-valued because an entity may or may not have a phone number. Attributes which are not null type must not accept a missing value.

On the basis of the multiplicity of values taken attributes can be classified either as Single-valued or Multi-valued. A single-valued attribute takes only one value for a given entity while a multi-valued attribute can take more than one value for a given entity. For instance, for the ongoing banking system, the attribute BranchManagerId is a single-valued while PhoneNumber

is a multi-valued attribute – i.e., one instance of a branch manager will have a single value for BranchManagerId but can have more than one phone number.

An attribute can be Derivable or Non-Derivable. Derivable attribute can be obtained by subjecting other attributes to computation. For instance, Age is a derived attribute for an AccountHolder entity because its value can be obtained by computing the difference between the current date and the DateOfBirth attribute.

Notes Since the value of the derived attribute can be computed from other attributes its value need not be stored in the database.

Self Assessment

Fill in the blanks:

10. On the basis of whether an attribute must take a value for an instance of a given entity or not, attributes can be either……..
11. On the basis of the ……. of values taken attributes can be classified either as Single-valued or Multi-valued.
12. ……. attribute can be obtained by subjecting other attributes to computation.

14.4 Relationships and Relationship Set

Entities do not exist in isolation form each other. They are more often than not related somehow with each other. A relationship is an association between entities. Let there be n number of entities given by: $E_1, E_2, E_3, \ldots, E_n$

A relationship is defined as a collection of attributes $(r_1, r_2, r_3, \ldots r_n)$ such that $r_1 \in E_1$ and $r_2 \in E_2$ and so on. A relationship set is a set of relationships of the same type.

Example: Consider the two entity sets – AccountHolders and Account. These two entities are related to each other because an Account is owned by an Account holder. We can give a name to this relationship – say, Owns. The relationship would read as:

- AccountHolder(Name="Eknath", Address="A-45, Tagore Garden", DateOfBirth="13/11/1966", DateOfOpeningAccount="12/10/2005") Owns Account (AccountNumber = "1345", OpeningBalance=50000)
- In fact another relationship may also exist in the opposite manner. In our case the relationship – Belongs To - can be thought of a relationship from Account entity to AccountHolder entity as:
- Account (AccountNumber = "1345", OpeningBalance=50000) Belongs To AccountHolder(Name="Eknath", Address="A-45, Tagore Garden", DateOfBirth="13/11/1966", DateOfOpeningAccount="12/10/2005")

Thus, the two relationships "Owns" and "Belongs To" are inversely related to each other. A relationship that exists between two entities is called binary relationship. But a relationship may exist between more than two entities as well. A relationship between three entity sets is called ternary relationship, and so on. The number of sets forming a relationship is known as degree of relationship. Therefore, a binary relationship is of degree 2 while ternary is that of 3 and a quinary of degree 5.

Notes

The objective of the E-R designing activity is to obtain these components of the E-R model.

E-R diagrams are very useful tools to develop and describe the E-R model of a database. In its original form ER diagrams were proposed by Dr. Pin-Shan Chen in the year 1976. The popularity of the tools caused many variants come up since then.

E-R diagrams are composed of symbolic notations for entities, attributes and relationships along with other aspects of these. A few of the symbolic notations and their meaning are listed in the Table 14.1.

Table 14.1: Symbols and their meaning used in ER-diagrams

Symbol	Meaning
	Entity
	Single-valued attribute
	Relationship
	Connector
	Derived attribute
	Multi-valued attribute
	Existence dependency
	Identifying relationship

Did u know? **What are the basic elements of ER Models?**

There are three basic elements in ER models:

1. Entities are the "things" about which we seek information.
2. Attributes are the data we collect about the entities.
3. Relationships provide the structure needed to draw information from multiple entities.

This list is not complete. Other symbols will be covered later. Using these symbols one can draw an entity-relationship diagram for a database. ER diagram for the entity AccountHolder is shown in figure 14.5(a) and for entity Account is shown in figure 14.5(b).

Notes

Figure 14.5(a): ER diagram for the entity: AccountHolder (Name, Address, DateOfBirth, DateOfOpeningAccount)

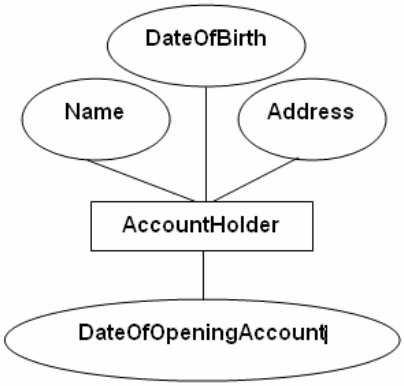
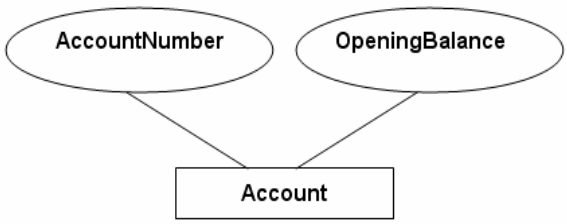
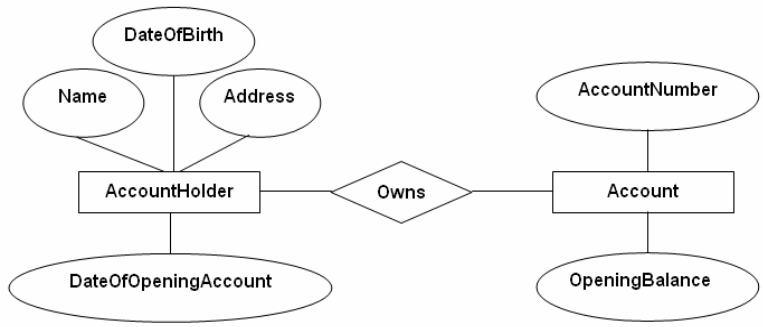


Figure 14.5(b): ER diagram for the entity: Account(AccountNumber, OpeningBalance)



The ER diagram for the relationship "Owns" between these two entities is shown in the figure 14.6.

Figure 14.6: ER diagram the relationship "Owns"



Self Assessment

Fill in the blanks:

1. E-R diagrams are very useful tools to develop and describe the …… of a database.
2. A relationship that exists between two entities is called …… relationship.

Notes

14.5 Cardinality and Modality

Constraints are domain rules that must be followed by the ER components for which they are defined. A common constraint laid on relationship is that of cardinality.

Task Cardinality constraint specifies the multiplicity of instances of entities related to each other by a relationship. Analyze

The relationship between entity sets may be many-to-many (M: N), one-to-many (1:M), many-to-one (M: 1) or one-to-one (1:1). The 1:1 relationship between entity sets E1 and E2 indicates that for each entity in either set there is at most one entity in the second set that is associated with it. The 1:M relationship from entity set E1 to E2 indicates that for an occurrence of the entity from the set E1, there could be zero, one, or more entities from the entity set E2 associated with it. Each entity in E2 is associated with at most one entity in the entity set E1. In the M: N relationship between entity sets E1 and E2, there is no restriction to the number of entities in one set associated with an entity in the other set. The database structure, employing the E-R model is usually shown pictorially using entity-relationship (E-R) diagram. The cardinality of a relation may be represented by a set of pictorial notations called Crow Feet notation shown listed in table 14.2. These symbols are paired to form desired cardinality.

Table 14.2: Crow feet notation

	Crow Feet Notation	Meaning
0		Zero
1		Exactly one
M		Many
Paired Notation		
	Paired Symbol	Meaning
0/1		Zeor or one
0/M		Zero or one to zero or one
1/M		Zero or one to exactly one

The "Crow's Foot" notation represents relationships with connecting lines between entities, and pairs of symbols at the ends of those lines to represent the cardinality of the relationship.

To illustrate these different types of relationships consider the following entity sets: DEPARTMENT, MANAGER, EMPLOYEE, and PROJECT.

The relationship between a DEPARTMENT and a MANAGER is usually one-to-one; there is only one manager per department and a manager manages only one department. This relationship between entities is shown in Figure 14.7. Each entity is represented by a rectangle and the relationship between them is indicated by a direct line. The relationship for MANAGER to DEPARTMENT and from DEPARTMENT to MANAGER is both 1:1.

Notes

Notes A one-to-one relationship between two entity sets does not imply that for an occurrence of an entity from one set at any time there must be an occurrence of an entity in the other set. In the case of an organization, there could be times when a department is without a manager or when an employee who is classified as a manager may be without a department to manage.

Figure 14.8 shows some instances of one-to-one relationships between the entities DEPARTMENT and MANAGER.

Figure 14.7: One-to-One Relationship

Figure 14.8: An instance of One-to-One Relationship

Following are the examples of the same relationship under different cardinality constraints (figure 14.9).

Figure 14.9: (a), (b), (c), (d), (e) and (f)

(a) A Department is managed by no Manager

(b) One Department is managed by exactly one Manager

(c) One Department is managed by many Managers

(d) One Department is managed by one or more Managers

Contd...

Notes

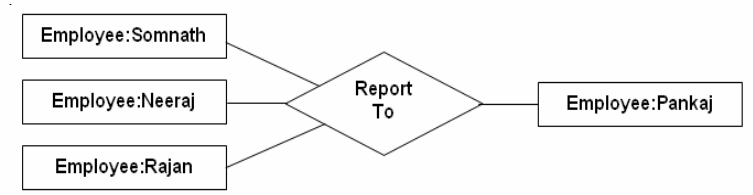
(e) A Department has none or exactly one Manager

(f) A Department has none or many Managers

A one-to-many relationship exists from the entity MANAGER to the entity EMPLOYEE because there are several employees reporting to the manager. As we just pointed out, there could be an occurrence of the entity type MANAGER having zero occurrences of the entity type EMPLOYEE reporting to him or her. A reverse relationship, from EMPLOYEE to MANAGER, would be many to one, since many employees may be supervised by a single manager. However, given an instance of the entity set EMPLOYEE, there could be only one instance of the entity set MANAGER to whom that employee reports (assuming that no employee reports to more than one manager). The relationship between entities is illustrated in Figure 14.10.

Figure 14.10 (a) and (b)

(a) (M:1) Many Employees Report To one Manager



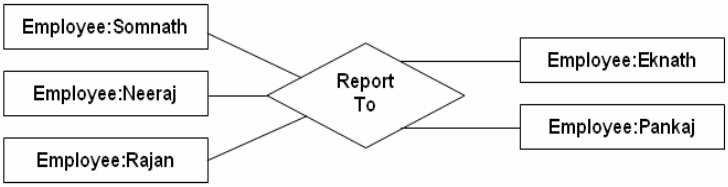
(b) An instance of (M:1) Relation

The relationship between the entity EMPLOYEE and the entity PROJECT can be derived as follows: Each employee could be involved in a number of different projects, and a number of employees could be working on a given project. This relationship between EMPLOYEE and PROJECT is many-to-many. It is illustrated in Figure 14.11.

Notes

Figure 14.11 (a) and (b)

(a): (M:N) Many Employees Report To many Managers



(b): An instance of (M:N) Relationship

Self Assessment

Fill in the blank:

15. Constraints are domain rules that must be followed by the …….. for which they are defined.

Caselet

Cloud Computing for Banking

Computing through the cloud offers new ways of doing business. Choosing a private, public, or a hybrid of private, public and community clouds should create new business models to conduct business in a highly competitive finance industry. Internet-based cloud IT solutions provide not only enormous cost reduction in infrastructure and operating expenditure, but also enable flexibility, agility and ability to scale up or down as per the needs of a business and its affordability.

Banking Sector

Emerging cloud services for banks aim at enhancing productivity by facilitating real time collaboration, being omnipresent and providing virtualisation. Cloud-based services help in knowing the customers' preference by social networking interfaces and focussing on better customer relations, human relations and finance management, helping the banks to retain customers and attract new consumers. Front-end bank offices connected to cloud-based back-end computing and analytics save substantial cost on licensing, energy and space.

Cloud based e-invoicing provides dynamic invoices, making payments when exchange rates are most favourable, and enabling the banks to network constantly with postal and telecommunications companies. It improves access to social network profiles to help reach out to new consumers in new markets.

Security Concerns

The banking industry is not so enthusiastic to embrace cloud computing in spite of its vast potential, because of the industry's concern on security, privacy, confidentiality, data

Contd...

Notes

integrity, and authentication requirements, along with location of data, availability, and recoverability.

Moreover, the industry has unique and dynamic regulatory, legal and compliance issues to address before switching to cloud services. Bankers apprehend that computing in "the cloud" is risky, as it involves outsourcing the data of its customers to third-party cloud service operators.

In order to take advantage of the emerging powerful Internet-based business solutions, what is needed is an IT technology architecture that combines the merits of the public cloud with the security and data-privacy of the private cloud.

The appropriate business strategy seems to be to outsource relatively less sensitive data to the public cloud infrastructure service with cryptography and simple password access, along with dedicated servers with firewalls and intrusion detection devices and other updated safety features for housing ultra-secure critical data centres that demand strong authentication for access.

Strategic Policy

There should be a clear strategic policy for cloud computing and management, prioritising data that can be entrusted to the cloud operator, with clearly defined Service Level Agreements (SLA) with milestones and a set time-frame, backed by a comprehensive governance structure.

While choosing the cloud service provider, it is important to look into the firm's financial stability, ability to improve functionality and service levels and integrate data across different technology platforms and cloud services.

The policy-based key management, with industry-standard encryption, is emerging as the cryptography model for better control on data in the cloud as the common encryption key management techniques are susceptible to vulnerability.

Cloud security services are emerging to address data security, privacy and compliance risks, as well as prevention of data theft, ensuring disaster management, and detecting compliance violations with robust server security for virtualised data centres.

14.6 Summary

To design any system, we require a mathematical model. Similarly, for DB Systems there are several models being used as given below:

- Relational Model
- Network Model
- Hierarchical Model

Out of all the above models, the Relational Model is used by most of the database management packages, including Oracle, Ingres, Sybase, etc.

14.7 Keywords

CODASYL/DBTG: (Conference on Data Systems Languages/Database Task Group) A standard specifying how to store and retrieve data to and from a network database.

Data Model: An abstract representation that captures the essence of relation with the data items.

Notes

Hierarchical Data Model: A data model that organizes data into a tree-like structure where each node has a single parent node, the root node being an exception in the sense that it does not have any parent.

Hierarchical Model: It is same as network model except that the records are rearrange in collection of trees rather then arbitrary graph.

Network Data Model: A data model that views database as group of related items associated with each other through links.

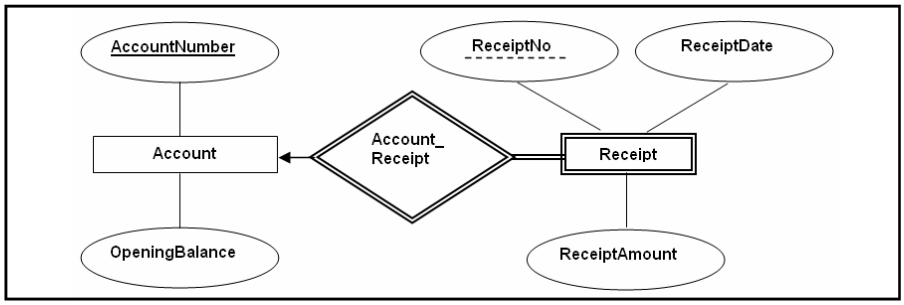
Network Model: In this model data is represented by collection of records and relationship with links which can be thought of as pointers.

Physical Data Model: These are used to describe data at lowest level in contrast to logical data model there are few physical model in use to of widely known once are unifying models and the frame memory model.

Relational Model: This uses a collection of tables to represent both data and the relationship among those data, each table has multiple column and each column has a unique name.

14.8 Review Questions

1. Describe the limitation that necessitated the design of different data models.
2. Compare and contrast between different data models.
3. Discuss the advantages and disadvantages of different data modeling techniques.
4. Differentiate between the network and hierarchical data models.
5. Write the following commands for the network data model.
   (a) Insert an ID (=110) as a child of node (ID=103)
   (b) Delete the node with ID(=008)
6. Convert the following E-R diagram into a structure of network data model.



7. Why do we require a mathematical model to design any system? Explain with examples.
8. What are the various techniques used to model data structure?
9. A hierarchical data model is a data model in which the data is organized into a tree-like structure. Explain.
10. Multiple Choice Questions:
    (i) A top-do-bottom relationship among the items in a database is established by a
    (a) Hierarchical schema (b) Network schema
    (c) Relational schema (d) All of the above

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-300360
Fax.: +91-1824-506111
Email: odl@lpu.co.in

978-81-946129-0-2
9 788194 612902

Notes

(ii) A network schema

(a) restricts the structure to a one-to-many relationship

(b) permits many-to-many relationships

(c) stores data in tables

(d) none of the above

(iii) In a relational schema, each tuple is divided into fields called

(a) Relations (b) Domains

(c) Queries (d) None of the above

(iv) Which of the following is not a logical database structure?

(a) tree (b) relational

(c) network (d) chain

(v) The relational model uses some unfamiliar terminology. A tuple is equivalent to a

(a) network (b) field

(c) file (d) data base

Answers: Self Assessment

1. data values
2. data model
3. relation
4. tuple
5. network data model
6. entity
7. attributes
8. graphically
9. physical
10. Null-valued or Non-Null-valued
11. Multiplicity
12. Derivable
13. E-R model
14. Binary
15. ER components

14.9 Further Readings

Books

Elmasari Navathe, *Fundamentals of Database Systems*, Third Edition, Pearson Education Asia

Peter Rob, Carlos Coronel, *Database Systems: Design, Implementation, and Management*, Seventh Edition, Thomson Learning

Raghu Ramakrishnan, Johannes Gehrke, *Database Management Systems*, Third Edition, McGraw-Hill, Higher Education

Silberschatz, Korth, Sudarshan, *Database System Concepts*, Fourth Edition, McGraw-Hill

Online links

http://en.wikipedia.org/wiki/Data_modeling

http://en.wikipedia.org/wiki/Data_model