



S.Y.B.Sc. (I.T)
SEMESTER - IV (CBCS)
SOFTWARE ENGINEERING
SUBJECT CODE: USIT404

Dr. Suhas Pednekar

Vice Chancellor
University of Mumbai, Mumbai

Prof. Ravindra D. Kulkarni

Pro Vice-Chancellor,
University of Mumbai

Prof. Prakash Mahanwar

Director,
IDOL, University of Mumbai

Programme Co-ordinator

: Shri Mandar Bhanushe

Head, Faculty of Science and Technology
IDOL, University of Mumbai – 400098.

Course Co-ordinator

: Ms. Gouri S.Sawant

Assistant Professor B.Sc.I.T, IDOL,
University of Mumbai- 400098.

Editor

: Dr Sujatha Iyer

Assistant Professor,
Satish Pradhan Dnyanasadhana College,
Thane.

Course Writers

: Dr R. Jayakarthish

Assistant Professor,
Vels Institute of Science Technology and
Advanced Studies.

: Dr Sumathi A

Assistant Professor
Computer science, Dr.N.G.P Arts
and science College, Coimbatore,
Tamil Nadu

: Ms. Prerna Satav

Assistant Professor,
KLE College Kalamboli.

: Ms. Preeti Bharanuke

Assistant Professor, M.SC.IT,
IDOL, University of Mumbai, Mumbai 98.

: Ms. Gouri S.Sawant

Assistant Professor B.Sc.I.T, IDOL,
University of Mumbai- 400098.

January 2022, Print - 1

Published by : Director,

Institute of Distance and Open Learning,
University of Mumbai,
Vidyanagari, Mumbai - 400 098.

DTP Composed and Printed by:

Mumbai University Press
Vidyanagari, Santacruz (E), Mumbai-400098.

CONTENTS

Unit/Chapter No.	Title	Page No.
Unit 1		
1.	An Introduction	01
2.	Software Requirements	28
3.	Software Processes	45
4.	Software Development Process Models	59
5	Agile Software Development	73
Unit 2		
6.	Socio-Technical System	89
7.	Critical Systems	108
8.	Requirement Engineering Process	124
9.	System Models.....	139
Unit 3		
10.	Architectural Design.....	155
11.	User Interface Design – Rapid Software Development	171
12.	Project Management	213
13.	Quality Management.....	229
Unit 4		
14.	Verification And Validation	245
15.	Software Measurement.....	260
16.	Software Cost Estimation.....	278
Unit 5		
17.	Process Improvement.....	307
18.	Service Oriented Software Engineering	317
19.	Software Reuse	339
20.	Distributed Software Engineering.....	363

S.Y.B.SC. (I.T)

SEMESTER - 4

Software Engineering

SYLLABUS

Unit	Details	Lectures
I	Introduction: What is software engineering? Software Development Life Cycle, Requirements Analysis, Software Design, Coding, Testing, Maintenance etc. Software Requirements: Functional and Non-functional requirements, User Requirements, System Requirements, Interface Specification, Documentation of the software requirements. Software Processes: Process and Project, Component Software Processes. Software Development Process Models. • Waterfall Model. • Prototyping. • Iterative Development. • Rational Unified Process. • The RAD Model • Time boxing Model. Agile software development: Agile methods, Plan-driven and agile development, Extreme programming, Agile project management, Scaling agile methods.	12
II	Socio-technical system: Essential characteristics of socio technical systems, Emergent System Properties, Systems Engineering, Components of system such as organization, people and computers, Dealing Legacy Systems. Critical system: Types of critical system, A simple safety critical system, Dependability of a system, Availability and Reliability, Safety and Security of Software systems. Requirements Engineering Processes: Feasibility study, Requirements elicitation and analysis, Requirements Validations, Requirements Management. System Models: Models and its types, Context Models, Behavioural Models, Data Models, Object Models, Structured Methods.	12
III	Architectural Design: Architectural Design Decisions, System Organisation, Modular Decomposition Styles, Control Styles, Reference Architectures.	12

	User Interface Design: Need of UI design, Design issues, The UI design Process, User analysis, User Interface Prototyping, Interface Evaluation. Project Management Software Project Management, Management activities, Project Planning, Project Scheduling, Risk Management. Quality Management: Process and Product Quality, Quality assurance and Standards, Quality Planning, Quality Control, Software Measurement and Metrics.	
IV	Verification and Validation: Planning Verification and Validation, Software Inspections, Automated Static Analysis, Verification and Formal Methods. Software Testing: System Testing, Component Testing, Test Case Design, Test Automation. Software Measurement: Size-Oriented Metrics, Function-Oriented Metrics, Extended Function Point Metrics Software Cost Estimation: Software Productivity, Estimation Techniques, Algorithmic Cost Modelling, Project Duration and Staffing	12
V	Process Improvement: Process and product quality, Process Classification, Process Measurement, Process Analysis and Modeling, Process Change, The CMMI Process Improvement Framework. Service Oriented Software Engineering: Services as reusable components, Service Engineering, Software Development with Services. Software reuse: The reuse landscape, Application frameworks, Software product lines, COTS product reuse. Distributed software engineering: Distributed systems issues, Client–server computing, Architectural patterns for distributed systems, Software as a service	12

Books and References:					
Sr. No.	Title	Author/s	Publisher	Edition	Year
1.	Software Engineering, edition,	Ian Somerville	Pearson Education.	Ninth	
2.	Software Engineering	Pankaj Jalote	Narosa Publication		
3.	Software engineering, a practitioner's approach	Roger Pressman	Tata Mcgraw-hill	Seventh	
4.	Software Engineering principles and practice	WS Jawadekar	Tata Mcgraw-hill		
5.	Software Engineering- A Concise Study	S.A Kelkar	PHI India.		
	Concept and Applications	Datta	Education		
7.	Software Design	D.Budgen	Pearson education	2nd	
8.	Software Engineering	KL James	PHI	EEE	2009

AN INTRODUCTION

Unit Structure

- 1.1 Objective
- 1.2 Introduction
- 1.3 Why is the term software really necessary?
- 1.4. What is software engineering?
- 1.5 Software Engineering in brief.
- 1.6 Tasks of software engineering
- 1.7. Life cycle Software engineering
 - 1.7.1 Classical Model waterfall
 - 1.7.2 Rapid Prototype model
 - 1.7.3 Spiral Model
 - 1.7.4 Market-driven Model of software engineering
 - 1.7.5 Open source model of software engineering
 - 1.7.6 Agile Programming Model of software engineering
- 1.8 Software Engineering Development Phase
 - 8.1 Requirement Analysis
 - 8.2 The feasibility study
 - 8.3 Design
 - 8.4 Coding
 - 8.5 Testing
 - 8.6 Maintnance
- 1.9 Summary
- 1.10 Questions
- 1.11 References

1.1 Objective

This chapter is about software engineering and why software engineering?

How software engineering can create, maintain, research and design all kinds of software applications to operating systems.

We can learn and understand the needs of software engineering from this chapter. Here we will discuss the goals and responsibilities of members of the group or team.

1.2 Introduction

Software engineering was 50-60 years old. At the North Atlantic Treaty Organization conference to discuss development issues of software. The word software engineering was introduced or became in 1969.

At that time extensive software systems were a delay and as per the user requirements, functionality did not provide, was unexpectedly expensive, and was unreliable.

During the 1970s and in the 1980s, the history of the programming is, PCs were simply starting to be accessible at a sensible expense.

There were numerous PC magazine's accessible at newspaper kiosks and book shops; these magazines were loaded up with articles depicting how to decide the substance of explicit memory areas utilized by PC working frameworks.

Different articles represent the calculations and their execution in some of the BASIC programming language.

Media inclusion recommended that the opportunities for developers were unlimited. **It appeared to be probable that the computerization of society and the major changes brought about by this computerization were driven by the activities of countless autonomously working developers.**

It appeared to be reasonable that the computerization of society and the principal changes brought about by this computerization were driven by the activities of an enormous number of autonomously working developers.

Nonetheless, one more pattern was happening, to a great extent stowed away from general visibility. Programming was filling incredibly in size and turning out to be amazingly perplexing. The development of word handling programming is a decent delineation.

In the last part of the 1970s, programming, for example, Microsoft Word and WordStar ran effectively on little PCs with just 64 kilobytes of client memory.

1.3 Why is the term software really necessary?

You may plan to be an engineer of independent applications for, say, cell phones, and keep thinking about whether this custom is essential. Here is a portion of the real factors. There are over 1,000,000 applications for iPhones and iPads on the App Store and thus, it tends to be difficult to have individuals find your application. You might have caught wind of the software engineer who created an application that sold 17,000 units in a single day and quit his place of employment to be a full-time application engineer.

Maybe you caught wind of Nick D'Aloisio, a British young person who sold an application named Summly to Yahoo! The cost was \$30 million, which is a considerable measure of cash for somebody still in secondary school.

(see the article <http://articles.latimes.com/2013/mar/26/business/la-fi-teen-millionaire-20130326>) computer programming is the term used to portray programming improvement that follows these standards.

In particular, the term computer programming alludes to a deliberate methodology that is utilized in the setting of a by and large acknowledged arrangement of objectives for the examination, plan, execution, testing, and support of programming.

The product delivered ought to be effective, dependable, usable, modifiable, compact, testable, reusable, viable, interoperable, and right. These terms allude both to frameworks and their parts. Large numbers of the terms are simple; nonetheless, we incorporate their definitions for fulfillment. You ought to allude to the as of late removed, yet at the same time accessible and helpful, IEEE standard glossary of PC terms (IEEE, 1990) or on the other hand to the "**Programming Body of Knowledge**" (SWEBOK, 2013) for related definitions.

1. **Effectiveness**—The product is delivered in the normal time and inside the constraints of the accessible assets. The product that is created runs inside the time expected for different calculations to be finished.
2. **Dependability**—The product proceeds true to form. In multiuser frameworks, the framework plays out its capacities even with other burdens on the framework.
3. **Ease of use**—The product can be utilized appropriately. This for the most part alludes to the convenience of the UI yet additionally concerns the materialism of the product to both the PC's working framework and utility capacities and the application climate.
4. **Modifiability**—The product can be effectively changed if the prerequisites of the framework change.
5. **Versatility**—The product framework can be ported to different PCs or frameworks without significant revising of the product. Programming that needs just to be recompiled to having an appropriately working framework on the new machine is viewed as truly compact.
6. **Testability**—The product can be effectively tried. This by and large implies that the product is written in a particular way.
7. **Reusability**—Some or the entirety of the product can be utilized again in different tasks. This implies that the product is secluded, that every individual programming module has an obvious interface, and that every individual

module has a characterized result from its execution. This frequently implies that there is a considerable degree of deliberation and over-simplification in the modules that will be reused regularly.

8. **Viability**—The product can be handily perceived and changed over the long haul if issues happen. This term is regularly used to portray the lifetime of enduring frameworks such as the aviation authority framework that should work for decades.
9. **Maintainability**—The software can be easily understood and changed over time if problems occur. This term is often used to describe the lifetime of long-lived systems such as the air traffic control system that must operate for decades.
10. **Interoperability**—The software system can interact properly with other systems. This can apply to software on a single, stand-alone computer or to software that is used on a network.
11. **Correctness**—The program produces the correct output.

1.4 What is software engineering?

1. Software engineering is the term which is composed by two word.
2. Programming is something other than a program code. A program is an executable code, which fills some computational need. Programming is viewed as assortment of executable programming code, related libraries and documentations. Programming, when made for a particular necessity is called programming item.
3. Designing, then again, is tied in with creating items, utilizing clear cut, logical standards and strategies
4. Software engineering is defined as a process of breaking down client prerequisites and afterward planning, building, and testing programming applications that will fulfill those necessities.
5. IEEE, in its standard 610.12-1990, defines software engineering as the application of a systematics, disciplined, which is a computable approach for the development, operation, and maintenance of software.
6. It was in the last part of the 1960s when numerous product projects fizzled.
7. Numerous products became over spending plan. Yield was an inconsistent programming that is costly to keep up with.
8. Bigger programming was troublesome and very costly to keep up with Bunches of programming can't fulfill the developing necessities of the client.
9. Intricacies of programming projects expanded at whatever point its equipment ability expanded. Interest for new programming expanded quicker contrasted with the capacity with create new programming.

10. Many individuals feel that product is just one more word for PC programs. In any case, when we are discussing computer programming, programming isn't only the
11. It might incorporate framework documentation, which depicts the construction of the framework; client documentation, which discloses how to utilize the framework, and sites for clients to download late item data.
12. This is one of the basic separations among expert and novice programming progress. In case you are making a program for yourself, no other person will utilize it
13. Furthermore, you don't need to stress over making program guides, recording the program plan, and so on in any case, assuming Instance you are making programming that others will utilize and different specialists will change then you normally need to give extra data like the code of the program

Questions	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market. Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable, and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation, and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.

Questions	Answer
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software, and process engineering. Software engineering is part of this more general process
What are the costs of software engineering?.	Roughly 60% of software costs are development costs; 40% are testing costs. For custom software, Evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of systems. For example, games should always be developed using a series of prototypes whereas safety-critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another
What differences has the Web made to software engineering?	The Web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse

Figure 1.1 frequently asked questions (**Reference by:” Software Engineering”, Ninth edition, Ian Sommerville**)

Software engineers are worried about creating programming items (i.e., programming which can be offered to a client). There are two sorts of programming items.

1. Generic products:

These are autonomous systems that are made by an improvement affiliation and sold on the open market to any customer who can Instance of this sort of item incorporate programming for PCs, for example, data sets, word processors, drawing bundles, and undertaking the board instruments.

It additionally incorporates supposed vertical applications intended for some particular reason, for example, library data frameworks, bookkeeping frameworks, or frameworks for keeping up with dental records.

Instances of this kind of item incorporate programming for PCs, for example, Information bases, word processors, drawing bundles, and undertaking the executive's instruments.

It likewise incorporates supposed vertical applications intended for some particular reason, for example, library data frameworks, bookkeeping frameworks, or frameworks for keeping up with dental records.

Customized (or bespoke) products:

These are frameworks that are dispatched by a specific client. For that client, A product worker for hire fosters the product, particularly for that client. Instances of this sort of programming incorporate control frameworks for electronic gadgets, frameworks written to help a specific business measure, and airport regulation frameworks.

A significant contrast between these sorts of programming is that, in nonexclusive items,

the association that fosters the product controls the product in particular. For custom items, the detail is generally evolved and constrained by the association that is purchasing the product. The product designers should work to that detail. Notwithstanding, the qualification between these framework item types is becoming progressively obscured. An ever-increasing number of frameworks are currently being worked with a conventional item as a base, which is then adjusted to suit the prerequisites of a client.

Undertaking Resource Planning (ERP) frameworks, like the SAP framework, are awesome instances of this methodology. Here, an enormous and complex framework is adjusted for an organization by consolidating data about business rules and cycles, reports required, etc.

At the point when we talk about the nature of expert programming, we need to take into account that the product is utilized and changed by individuals separated from its designers. Quality is in this manner not simply worried about what the product does of the framework programs and related documentation. This is reflected in supposed quality or non-practical programming credits. Instances of these characteristics are the product's reaction time to a client inquiry and the understandability of the program code.

The particular arrangement of qualities that you may anticipate from a product framework relies upon its application. In this manner, a financial framework should be secure, and the intuitive game should be responsive, a phone exchanging framework should be dependable, etc. These can be summed up into the arrangement of characteristics.

1.5 Software engineering:

Computer programming is a designing discipline that is worried about all parts of programming creation from the beginning phases of framework determination through to keeping up with the framework after it has gone into utilization. In this definition, there are two key expressions:

1. **Designing discipline** Engineers make things work. They apply speculations, techniques, and apparatuses where these are suitable. In any case, they use them specifically and consistently attempt to find answers for issues in any event, when there are no appropriate speculations and strategies.

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency, therefore, includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use

By Ref: Software engineering (9th addition) Ian Sommerville.

- I. Specialists likewise perceive that they should work to hierarchical and monetary limitations so they search for arrangements inside these limitations.
- II. All parts of programming creation Software designing aren't recently concerned with the specialized cycles of programming advancement.
- III. It additionally incorporates exercises, for example, programming project the board and the advancement of apparatuses, strategies, what's more, hypotheses to help programming creation.

- IV. Designing is tied in with getting consequences of the necessary quality inside the timetable what's more, spending plan.
- V. This frequently includes making compromises—engineers can't be fussbudgets. Individuals composing programs for themselves, in any case, can invest as much energy as they wish on the program improvement.
- VI. As a rule, programmers embrace an orderly and coordinated way to deal with their work, as this is frequently the best method to create excellent programming.

In any case, designing is tied in with choosing the most proper strategy for a bunch of conditions so a more innovative, less proper way to deal with advancement might be powerful in certain conditions. Less conventional advancement is especially proper for the improvement of electronic frameworks, which requires a mix of programming

what's more, graphical plan abilities.

- **Computer programming is significant for two reasons:**

1. To an ever increasing extent, people and society depend on cutting edge programming frameworks. We should have the option to deliver solid and dependable frameworks financially and rapidly
2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of systems, the majority of costs are the costs of changing the software after it has gone into use.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. There are four fundamental activities that are common to all software processes.

- **These activities are:**

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires.
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

Different types of systems need different development processes. For example, real-time software in an aircraft has to be completely specified before development begins. In e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organized in different

ways and described at different levels of detail depending on the type of software being developed.

Software processes in more detail:

- **Software engineering is related to both computer science and systems engineering:**

1. **Computer science** is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers. Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science cannot always be applied to large, complex problems that require a software solution.
2. **System engineering** is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design, and system deployment, as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.)

As I discuss in the next section, there are many different types of software. There is no universal software engineering method or technique that is applicable for all of these.

- **However, there are three general issues that affect many different types of software:**

1. **Heterogeneity Increasingly**, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. As well as running on general-purpose computers, the software may also have to execute on mobile phones. You often have to integrate new software with older legacy systems

written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.

2. **Business and social change** Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and rapidly develop new software. Many traditional software engineering techniques are time-consuming and delivery of new systems often takes longer than planned. They need to evolve so that the time required for the software to deliver value to its customers is reduced.
3. **Security and trust** as software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. We have to make sure that malicious users cannot attack our software and that information security is maintained.

Of course, these are not independent issues. For example, it may be necessary to make rapid changes to a legacy system to provide it with a web service interface. To address these challenges, we will need new tools and techniques as well as innovative ways of combining and using existing software engineering methods.

1.6 There are several tasks that are part of every software engineering project:

- Analysis of the problem
- Determination of requirements
- Design of the software
- Coding of the software solution
- Testing and integration of the code
- Installation and delivery of the software
- Documentation
- Maintenance
- Quality assurance
- Training
- Resource estimation
- Project management

We will not describe either the analysis or training activities in this book in any detail.

Analysis of a problem is very often undertaken by experts in the particular area of application, although, as we shall see later, the analysis can often benefit from input from software engineers and a variety of potential users. Think of the problem that Apple's software designers faced when Apple decided to create its iOS operating system for smartphones using as many existing software components from the OS X operating system as possible.

Apple replaced a Linux-based file system where any running process with proper permissions can access any file to one where files are in the province of the particular

The application that created them. The changes in the user interface were also profound, using

the “**capacitive touch**” where a swipe of a single finger and a multi-finger touch are both parts

of the user interface.

We now briefly introduce the other activities necessary in software development

Unfortunately, a discussion of software training is beyond the scope of this chapter.

We now briefly introduce the other activities necessary in software development. Most of these activities will be described in detail in a separate chapter. You should note that these tasks are generally not performed in a vacuum. Instead, they are performed as part of an organized sequence of activities that is known as the “**software life cycle**”

1.7 Software Development life cycle :

1. It is the process to develop, design, and test high quality it is called the Software Development life cycle.
2. SDLC is a deliberate cycle (systematic process) for building programming that ensures the quality and rightness of the item manufactured.
3. SDLC process implies making high quality programming that meets customer presumptions or expectations.
4. The system development should be complete or finished within given period duration and cost.
5. SDLC consist of detailed of arrangements or plan which explains you how to plan, develop, build, maintain, replace specific software.
6. Every phase of the SDLC life Cycle has its own interaction and expectations that feed into the next stage.
7. SDLC is short form of Software Development Life Cycle and it is called as the Application Development Life-Cycle.

There are reasons for developing a software system why SDLC is important.

1. It offers a basis for project planning, scheduling, and estimating
2. Gives a structure to a standard arrangement of exercises and expectations. It is a system for project following and control.
3. Expands perceivability of undertaking wanting to all elaborate stakeholder of the improvement(development) process.
4. To improved client relations, increased and enhanced development speed.

We will speedily look at six fundamental models of **software development life cycles** in this reliable district.

1. Classical waterfall model
2. Rapid prototyping model
3. Spiral model
4. Market-driven model
5. Open source development model
6. Agile development model

(Only the first four life cycle models listed were described in the first edition of this book.

There is a seventh software development life cycle, called hardware and software concurrent development, or development, that is beyond the scope of this book.)

Each of these software life cycle models will be discussed in a separate section. For simplicity, we will leave documentation out of our discussion of the various models of software development. Keep in mind that the models discussed to describe a “pure process”; there are likely to be many variations on the processes described herein in almost any actual software development project.

- **Classical Waterfall Model**

One of the most common models of the software development process is called the classical waterfall model and is illustrated in Figure 1.3. This model is used almost exclusively in situations where there is a customer for whom the software is being created. The model is essentially never applied in situations where there is no known customer, as would be the case of software designed for the commercial market to multiple retail buyers.

In the simplest classical waterfall model, the different activities in the software life cycle are considered to occur sequentially. The only exception is that two activities that are adjacent in Figure 1 are allowed to communicate via feedback. This artificiality is a major reason that this model is no longer used extensively in industry. A second reason is that technology changes can

make the original set of requirements obsolete. Nonetheless, the model can be informative as an illustration of a software development life cycle model.

A life cycle model such as the one presented in Figure 1.3 illustrates the sequence of many of the major software development activities, at least at a high level. It is clear, for example, that specification of a system's requirements will occur before testing of individual modules. It is also clear from this abstraction of the software development life cycle

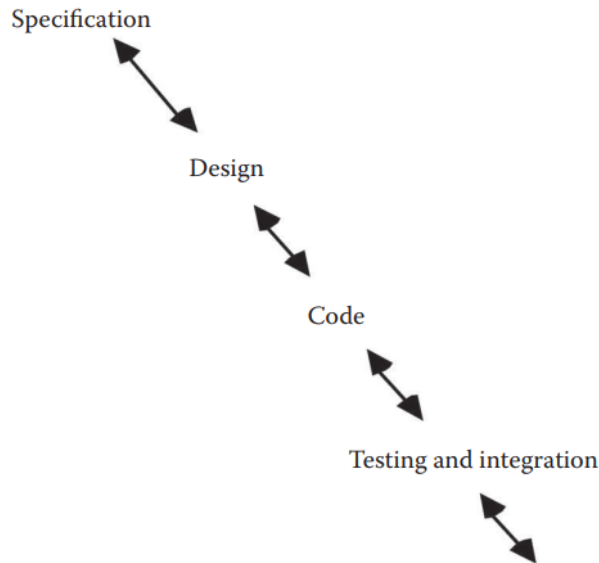


Fig 1

that there is feedback only between certain pairs of activities: requirements specification

and software design; design and implementation of source code; coding and testing; and

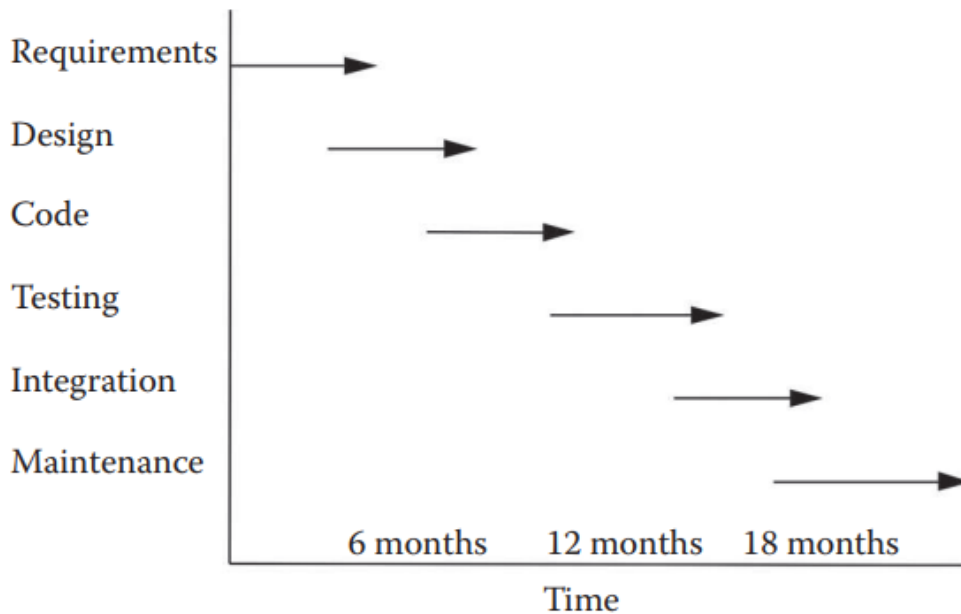
testing and maintenance.

What is not clear from this illustration is the relationship between the different activities and the time when one activity ends and the next one begins. This stylized description

of the classical waterfall model is usually augmented by a sequence of milestones in which

every two adjacent phases of the life cycle interface. A simple example of a milestone chart

for a software development process that is based on the classical waterfall model is given fig 2



An example of a milestone chart for a classical waterfall software development process.

final (or critical) requirements review. In nearly all organizations that use the classical waterfall model of software development, the action of having the requirements presented at the critical requirements review accepted by the designers and customer will mark the “official” end of the requirements phase.

There is one major consequence of a decision to use the classical waterfall model of software development. Any work by the software’s designers during the period between the preliminary and (accepted) final, or critical, requirements reviews may have to be redone if it is no longer consistent with any changes to the requirements. This forces the software development activities to essentially follow the sequence that was illustrated in Figure 1.3.

Note that there is nothing special about the boundary between the requirements specification and design phases of the classical waterfall model. The same holds true for the design and implementation phases, with system design being approved in a preliminary design review, an intermediate design review, and the final (or critical) design review. The pattern applies to the other interfaces between life cycle phases as well.

The classical waterfall model is very appealing for use in those software development projects in which the system requirements can be determined within a reasonable period and, in addition, these requirements are not likely to change very much during the development period. Of course, there are many situations in which these assumptions are not valid.

- **Rapid Prototyping Model**

Due to the artificiality of the classical waterfall model and the long lead time between setting requirements and delivering a product, many organizations follow the rapid prototyping or spiral models of software development. These models are iterative and require the creation of one or more prototypes as part of the software development process. The model can be applied whether or not there is a known customer. The rapid prototyping is illustrated in Figure 3 (The spiral model will be discussed).

The primary assumption of the rapid prototyping model of software development is that the complete requirements specification of the software is not known before the software

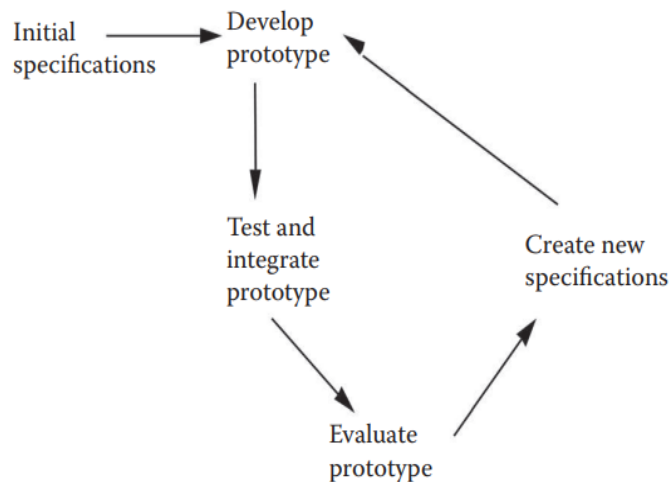


FIGURE 3 The rapid prototyping model of the software development life cycle is designed and implemented. Instead, the software is developed incrementally, with the specification developing along with the software itself. The central point of the rapid prototyping method is that the final software project produced is considered as the most acceptable of a sequence of software prototypes.

The rapid prototyping method will now be explained in more detail, illustrating the basic principles by the use of a system whose development history may be familiar to you: the initial development of the Microsoft Internet Explorer network browser. (The same issues apply to the development of the Mozilla Firefox, Apple Safari, and Google Chrome browsers, to a large extent.) Let us consider the environment in which Microsoft found itself.

For strategic reasons, the company decided to enter the field of Internet browsers.

The success of Mosaic and Netscape, among others, meant that the development path was constrained to some degree. There had to be support for HTML and most existing web pages had to be readable by the new browser. The user interface had to have many features with the same functionality as those of Netscape and Mosaic.

However, the user interfaces had to be substantially different from those systems in order to avoid copyright issues. Some of the new systems could be specified in advance. For example, the new system had to include a parser for HTML documents. In addition, the user interface had to be mouse-driven.

However, the system could not be specified fully in advance. The user interface would have to be carefully tested for usability. Several iterations of the software were expected before the user interface could be tested to the satisfaction of Microsoft. The users testing the software would be asked about the collection of features available with each version of Internet Explorer.

The user interface not only had to work correctly but the software had to have a set of features that were of sufficient perceived value to make other potential users change from their existing Internet browsers if they already used one or purchase Microsoft Internet Explorer if they had never used such a product before.

Thus, the process of development for this software had to be iterative. The software was developed as a series of prototypes, with the specifications of the individual prototypes changing in response to feedback from users.

Many of the iterations reflected small changes, whereas others responded to disruptive changes such as the mobile browsers that

work on smaller screens on smartphones and tablets with their own operating systems and

the need to interoperate with many Adobe products, for example

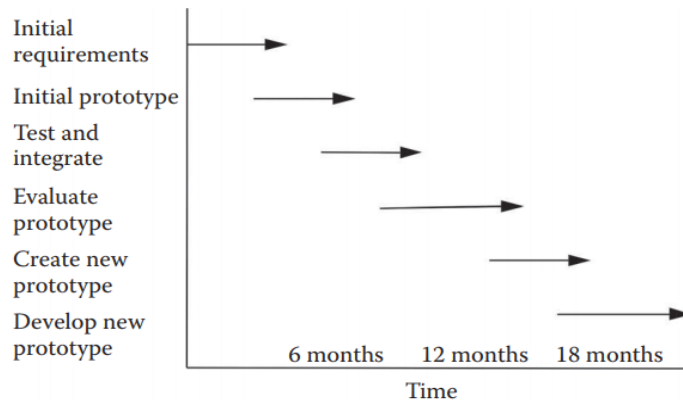
Now let us consider an update to this browser. There have been changing standards for HTML, including frames, cascading style sheets, a preference for the HTML directive `<p>` instead of `<div>`, and a much more semantically powerful version, HTML 6.

Here the browser competition is between IE and several browsers that became popular after IE became dominant, such as Mozilla's Firefox, Apple's Safari, and Google's Chrome.

Of course, there are many software packages and entire suites of tools for website development, far too many to mention. Anyone creating complex, graphics-intensive websites is well aware of the subtle differences between how these browsers display some types of Information.

- **Spiral Model**

Another iterative software development life cycle is the spiral model developed by Barry Boehm (Boehm, 1988). The spiral model differs from the rapid prototyping model primarily in the explicit emphasis on the assessment of software risk for each prototype during the evaluation period.



An example of a milestone chart for a rapid prototyping software development process.

The term risk is used to describe the potential for disaster in the software project. Note that this model is almost never applied to projects for which there is no known customer.

Clearly, there are several levels of disasters, and different organizations and projects may view identical situations differently. Examples of commonly occurring disastrous situations in software development projects include the following:

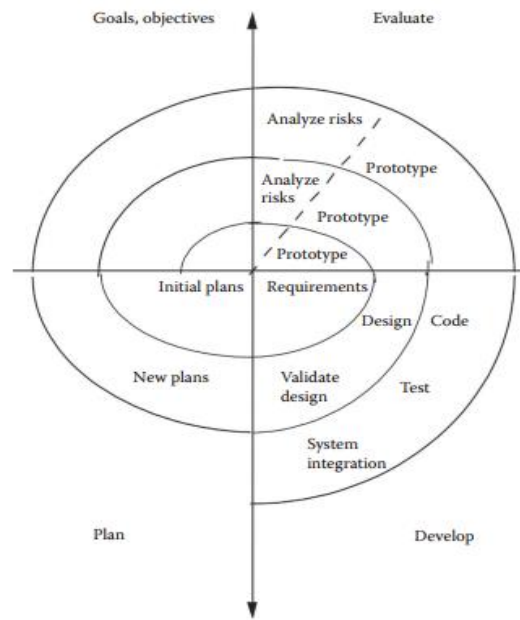
- The software development team is not able to produce the software within the allotted budget and schedule.
- The software development team is able to produce the software and is either over budget or schedule but within an allowable overrun (this may not always be disastrous).
- The software development team is not able to produce the software within anything resembling the allotted budget.
- After the considerable expenditure of time and resources, the software development

The team has determined that the software cannot be built to meet the presumed requirements at any cost.

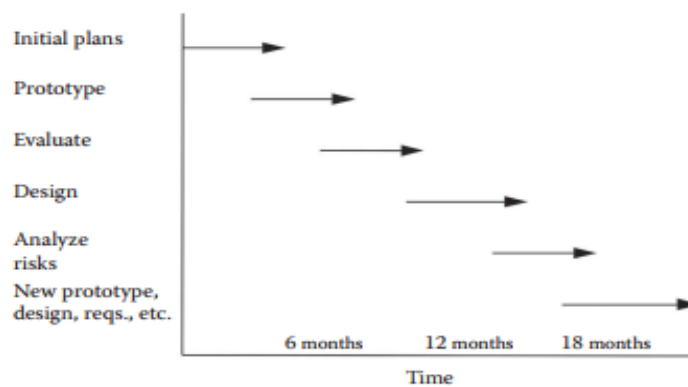
Planning is also part of the iterative process used in the spiral development model.

The classical waterfall and rapid prototyping software development models, a milestone chart can be used for the spiral development process.

rapid prototyping and spiral models are classified as iterative because there are several instances of intermediate software that can be evaluated before a final product is Produced.



The spiral model of the software development life cycle.



- ### Market-Driven Model of Software Development

It should be clear that none of the models of the software development life cycle previously described in this book are directly applicable to the modern development of software for the general consumer market.

The models previously discussed assumed that there was time for relatively complete initial requirements analysis (as in the classical waterfall model) or for an iterative analysis (as in the rapid prototyping and spiral models with their risk assessment and discarding of unsatisfactory prototypes).

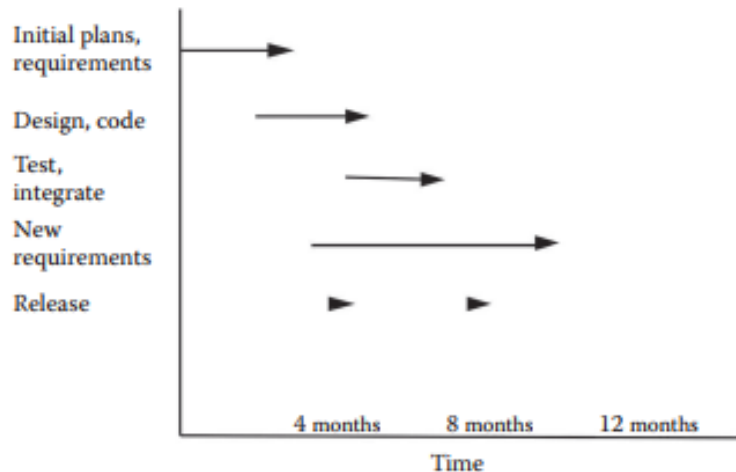
These models do not, for example, address the realities of the development of software for personal computers and various mobile devices. Here, the primary pressure driving the development process is getting a product to market with sufficient quality to satisfy consumers and enough desirable new features to maintain, and even increase market share.

This is a relatively new approach to software development and no precise, commonly accepted models of this type of software development process have been advanced as yet, at least not in the general literature.

The reality is that the marketing arm of the organization often drives the process by demanding that new features be added, even as the product nears its target release date.

Thus there is no concept of a “requirements freeze,” which was a common, unwritten assumption of all the previous models at various points.

We indicate the issues with this type of market-driven “concurrent engineering” in the stylized milestone chart in Figure



An example of a milestone chart for a market-based, concurrently engineered software development process.

The developer does not do so or is slow in making the changes, bad reviews result, and sales will plummet.

- **Open Source Model of Software Development**

Open-source software development is based on the fundamental belief that the intellectual content of software should be available to anyone for free. Hence, the source code for any project should be publicly available for anyone to make changes to it. As we will discuss later in this section, “free” does not mean “no cost,” but refers to the lack of proprietary ownership so that the software can be easily modified for any new purpose if necessary.

Making source code available is the easy part. The difficulty is in getting various source code components developed by a wide variety of often geographically separated people to be organized, classified, maintainable, and of sufficient quality to be of use to themselves and others. How is this done?

In order to answer this question, we will separate our discussion of open source projects into two primary categories: (1) projects such as new computer languages, compilers, software libraries, and utility functions that are created for the common good; and (2) reuse of the open-source versions of these computer languages, compilers, software libraries, and utility functions to create a new software system for a specific purpose.

Here is a brief overview of how many open source projects involving computer languages, compilers, software libraries, and utility functions appear to be organized. The primary model for this discussion of open-source software development is the work done by the Free Software Foundation, although there also are aspects of wiki-based software development in this discussion.

An authoritative person or group decides that a particular type of software application or utility should be created. Initial requirements are set and a relatively small number of software modules are developed. They are made publicly available to the open-source community by means of publication on a website, possibly one separate from the one that contains relatively complete projects.

The idea is that there is a community of developers who wish to make contributions to the project. These developers work on one or more relevant source code modules and test the modules' performance to their own satisfaction, then submit the modules for approval by the authoritative person or group. If the modules seem satisfactory, they are uploaded to a website where anyone can download them, read the source code, and test the code. Anyone can make comments about the code and anyone can change it

- **Agile Programming Model of Software Development**

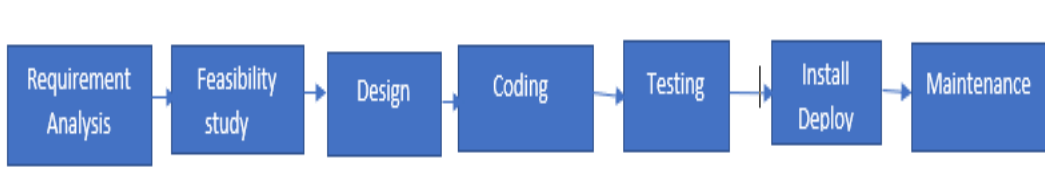
The roots of the concept of agile development can be traced back to at least as far as the “Skunk Works” efforts to engineer projects at the company now known as Lockheed Martin. (A Wikipedia article accessed May 5, 2015, states that the term Skunk Works “is an official alias for Lockheed Martin’s Advanced Development Programs [ADP], formerly called Lockheed Advanced Development Projects.”) Skunk Works came to the fore by the The early 1940s, during World War II.

The term is still used to describe both highly autonomous teams and their projects, which are usually highly advanced. Many of the Skunk Works projects made use of existing technologies, a forerunner for component-based software development, with the efficient use of existing software components that can be as large as entire subsystems.

The principles of what is now known as agile programming are best described by the “Agile Manifesto,” which can be found at <http://agilemanifesto.org/principles.html> and whose basic principles are listed here for convenience.

1.8 Software Development phases:

The whole SDLC measure is partitioned into the accompanying SDLC steps



1.8.1. The Requirement Analysis:

The requirement is the first stage in the SDLC process. It is conducted by the senior team members with inputs from all the stakeholders and domain experts in the industry. Planning for the quality assurance requirements and recognition of the risks involved is also done at this stage.

This stage gives a clearer picture of the scope of the entire project and the anticipated issues, opportunities, and directives that triggered the project.

Requirements Gathering stage needs teams to get detailed and precise requirements. This helps companies to finalize the necessary timeline to finish the work of that system.

1.8.2. The feasibility study:

When the prerequisite investigation stage is finished the following sdhc step is to characterize and archive programming needs. This cycle led with the assistance of the 'Programming Requirement Specification' archive otherwise called the 'SRS' report. It incorporates all that which ought to be planned and created during the task life cycle.

There are primarily five kinds of plausibility checks:

1. **Economical:** Can we finish the venture inside the spending plan or not?
2. **Legal:** Can we handle this venture as digital law and other administrative structures/compliances.
3. **Operational feasibility:** Can we make tasks which is normal by the customer?
4. **Technical:** Need to check whether the current PC framework can uphold the product

5. **Schedule:** Decide that the task can be finished inside the given timetable or not.

In this third stage, the framework and programming configuration records are ready according to the necessity particular report. This characterizes generally framework engineering.

1.8.3. Design

In this third phase, the system and software design documents are prepared as per the requirement specification document. This helps define the overall system architecture.

This design phase serves as input for the next phase of the model.

There are two kinds of design documents developed in this phase

Significant Level Design (HLD)

1. Brief depiction and name of every module
2. A layout about the usefulness of each module
3. Interface relationship and conditions between modules
4. Data set tables distinguished alongside their critical components
5. Complete design graphs alongside innovation subtleties

Low-Level Design (LLD)

8.The practical rationale of the modules Data set tables, which incorporate sort and size Complete detail of the interface Addresses a wide range of reliance issues Posting of mistake messages Complete info and yields for each module

1.8.4. Coding:

When the framework configuration stage is finished, the following stage is coding. In this stage, engineers begin to construct the whole framework by composing code utilizing the picked programming language. In the coding stage, undertakings are separated into units or modules and appointed to the different designers. It is the longest period of the Software Development Life Cycle measure.

In this stage, the Developer needs to adhere to certain predefined coding rules. They likewise need to utilize programming apparatuses like compilers, mediators, debuggers to create and execute the code.

1.8.5. Testing:

When the product is finished, and it is conveyed in the testing climate. The testing group begins testing the usefulness of the whole framework. This is done to check that the whole application works as indicated by the client prerequisite.

During this stage, QA and testing group might discover a few bugs/absconds which they convey to designers. The advancement group fixes the mess and sends back to QA for a re-test. This cycle proceeds until the product is sans bug, stable, and working as indicated by the business needs of that framework.

1.8.6. Installation and Deployment:

When the product testing stage is finished and no bugs or blunders are left in the framework then the last arrangement measure begins. In view of the criticism given by the venture chief, the last programming is delivered and checked for arrangement issues assuming any.

1.8.7. Maintenance:

When the framework is sent, and clients begin utilizing the created framework, the accompanying 3 exercises happen

Bug fixing – bugs are accounted for due to certain situations which are not tried by any stretch of the imagination

Overhaul – Upgrading the application to the fresher variants of the Software

Upgrade – Adding some new elements into the current programming

The fundamental focal point of this SDLC stage is to guarantee that necessities keep on being met and that the framework keeps on proceeding according to the particular referenced in the main stage.

1.9 Summary:

- Computer programming is a designing discipline that is worried about all parts of programming creation.
- Software isn't only a program or projects yet in addition incorporates documentation. Fundamental programming item ascribes are viability, constancy, security, effectiveness, and agreeableness.
- The product interaction incorporates the entirety of the exercises engaged with programming advancement. The high level exercises of determination, improvement, approval, and advancement are important for all product measures.
- The key ideas of programming are all around material to a wide range of framework advancement. These basics incorporate programming measures, constancy, security, prerequisites, and reuse.
- There are a wide range of sorts of frameworks and each requires proper computer programming devices and methods for their turn of events. There are scarcely any, particular plan and execution procedures that are relevant to a wide range of frameworks.

- The essential thoughts of programming are relevant to a wide range of programming frameworks.
- These essentials incorporate oversight programming measures, programming reliability and security, prerequisites designing, and programming reuse.
- Software engineers have liabilities to the designing calling and society. They ought to not just be worried about specialized issues.
- Professional social orders distribute sets of accepted rules that set out the principles of conduct anticipated of their individuals.

1.10 Questions:

1. Clarify why proficient programming isn't only the projects that are created for a client.
2. Why we need software engineering?
3. What is software engineering? Explain in brief.
4. Explain Software development Life Cycle.

1.11 References:

- Introduction to Software Engineering, Second Edition, Series Editor, Ronald J. Leach
- Software engineering, Ninth edition, Ian Sommerville.
- <https://www.guru99.com/software-development-life-cycle-tutorial.html>
- <http://imappl.org/~rjl/Software-Engineering>
- Ada, Reference Manual for the Ada Programming Language, ANSI-MIL-STD-1815A, 1983.
- Ada, Reference Manual for the Ada Programming Language, ANSI-MIL-STD-1815B, 1995.
- Ada, 2005 Language Reference Manual, 2005, <http://www.adaic.org/ada-resources/standards/ada05/>.
- Ada, 2012 Language Reference Manual, 2012, <http://www.ada-auth.org/standards/ada12.html>.
- Albrecht, A. J., Measuring application development productivity, Proceedings of the IBM Applications
- Development Joint SHARE/GUIDE Symposium, 83–92, Monterey, California, 1979.

- Albrecht, A. J., and Gaffney, Jr., J. E., Source lines of code, and development effort prediction: A software science validation, IEEE Transactions on Software Engineering, vol. SE-9, 639–648, 1983.
- American National Standards Institute, Standard Glossary of Software Engineering Terminology, ANSI/IEEE Standard 729–1991.
- American National Standards Institute, Recommended Practice for Software Reliability, ANSI Standard R-012-1992.
- Armour, P. G., The business of software: When faster is slower, Communications of the ACM, vol. 56, no. 10, 30–32, October 2013.
- Arnold, R. S., ed., Software Reengineering, IEEE Press, Los Alamitos, California, 1992.
- Arnold, R. S., and Bohner, S., Software Change Impact Analysis, IEEE Press, Los Alamitos, California, 1997.
- Arnold, T. R., and Fuson, W. A., Testing “in a perfect world,” Communications of the ACM, vol. 37, no. 9, 78–86, September 1994.
- Atkinson, C., Object-Oriented Reuse, Concurrency, and Distribution, ACM Press, New York, 1991.
- Babar, M. A., Kitchenham, B., Zhu, L., and Jeffery, R., An exploratory study of groupware support for distributed software architecture evaluation process, Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04), 222–229, November 30–December 3, 2004.
- Baker, A. L., and Zweben, S. H., The use of software science in evaluating modularity, IEEE Transactions on Software Engineering, vol. SE-5, no. 3, 110–120, 1979.
- Barbey, S., and Strohmeier, A., The problematics of testing object-oriented software, Proceedings of the Second Conference on Software Quality Management (SQM '94), Edinburgh, Scotland, vol. 2, 411–426, July 26–28, 1994.
- Barker, T. T., and Dragga, S., Writing Software Documentation: A Task-Oriented Approach, Allyn & Bacon, New York, 1997.
- Barnard, J., and Price, A., Managing code inspection information, IEEE Software, vol. 11, no. 2, 59–69, March 1994.
- Basili, V. R., and Selby, R. W., Comparing the effectiveness of software testing strategies, University of Maryland at College Park, Department of Computer Science, Technical Report Number TR-1501, 1985.

- Basili, V. R., and Rombach, H. D., The TAME project: Towards improvement-oriented software development, IEEE Transactions on Software Engineering, vol. SE-14, no. 6, 758–773, 1988.
- Basili, V. R., Viewing maintenance as reuse-oriented software development, IEEE Software, vol. 7, no. 1, 19–25, January 1990.
- Gotterbarn, D., Miller, K. and Rogerson, S. (1999). Software Engineering Code of Ethics is Approved. Comm. ACM, 42 (10), 102–7.
- Holdener, A. T. (2008). Ajax: The Definitive Guide. Sebastopol, Ca.: O'Reilly and Associates.
- Huff, C. and Martin, C. D. (1995). Computing Consequences: A Framework for Teaching Ethical Computing. Comm. ACM, 38 (12), 75–84.
- Johnson, D. G. (2001). Computer Ethics. Englewood Cliffs, NJ: Prentice Hall.
- Laudon, K. (1995). Ethical Concepts and Information Technology. Comm. ACM, 38 (12), 33–9.
- Naur, P. and Randell, B. (1969). Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany. 7th to 11th October 1968.

SOFTWARE REQUIREMENTS

Unit Structure

- 2.0 Objectives
- 2.1 Introduction
- 2.2 Functional and Non-Functional Requirements
- 2.3 User Requirements
- 2.4 System Requirements
- 2.5 Interface Specification
- 2.6 Documentation of the Software Requirements
- 2.7 Summary
- 2.8 References
- 2.9 Questions

2.0 Objectives

Following are the listed objectives of the chapter:

- know the ideas of user and system requirements and why these requirements need to be written in different ways
- understand the differences between functional and non-operational software requirements
- understand how the necessary conditions may be organized in a software specification for the document
- appreciate the principal specifications for the engineering activities of elicitation, analysis and validation, and the relationships between these activities
- understand why the terms and conditions the management is necessary and how it supports other standards laid down in engineering events

2.1 Introduction

Requirement's engineering is conceded with instituting what the system should do, its desired and essential emergent properties, and the restrictions on system operation and the software development processes. You can consequently think of

requirements engineering as the communications process between the software clients and customers and the software developers.

Requirement's engineering is not just a technical process. The system requirements are influenced by users' likes, dislikes, and prejudices, and depending on the political and organisational issues. These are fundamental human traits, and new technologies, such as use-cases, scenarios and proper methods do not help us much in solving these problems.

The term 'requirement' is not used regularly in the software industry. In some cases, a requirement is simply a high-level, abstract declaration of a service that a system should provide or a constraint on a system. Some of the problems that occur during the requirements engineering process are a result of failing to make a clear distinction between these different levels of description. There are two terminologies utilized in the requirement engineering User Requirements and System Requirements.

User requirements and system requirements may be defined as follows:

1. **User requirements** are statements, in a natural language plus diagrams, of what facilities the system is expected to provide to system users and the limitations under which it must operate.
2. **System requirements** are a more complete description of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define precisely what is to be implemented. It may be part of the agreement between the system buyer and the software developers.

Different levels of requirements are helpful because they communicate information about the system to the different kinds of reader example:

User Requirement Definition:

The chemist billing system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification:

- On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
- The system shall automatically generate the report for printing after 12.00 on the last business day of the month.

- A report shall be created for each clinic and shall make a list the individual drug names, the total number of prescriptions, the number of doses approved, and the total cost of the prescribed drugs.
- If drugs are available in different dose units (e.g., 10 mg, 20 mg) the individual reports shall be created for every dose unit.
- Access to all cost reports shall be limited to authorized users listed on a administration access control list.

2.2 Functional and non-functional requirements

Software system requirements are frequently categorized as functional requirements or non-functional requirements:

Functional requirements: The following are the statements of services the system should provide, how the system should react to inputs, and how the system should perform situations. In some cases, the functional requirements may also explicitly define what the system should not do. Functional system requirements vary from general specifications covering what the system should do in order to very particular requirements reflecting local ways of working or the organization's existing systems. For example, here are examples of functional requirements:

- 1) Authentication and authorization of user whenever he/she logs into the system.
- 2) System should raise query when hazardous issue is highlighted.
- 3) A Verification email and OTP is sent to user whenever he/she registers for the first time on some software system or during ecommerce transaction an SMS is sent for the amount deduction.

The functional requirements specification of a system must be both complete and consistent. Completeness means that all services required by the user must be defined. Consistency means that requirements should not have conflicting definitions.

In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness. One reason for this is that it is simple to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is an individual or a role that is affected by the system in some way. Stakeholders have different— and frequently inconsistent—needs. These inconsistencies may not be evident when the requirements are first specified, so conflicting requirements are included in the specification.

Non-functional requirements These are constraints on the services or features offered by the system. They include timing constraints, constraints on the developmental process, and constraints imposed by standards. Non-functional requirements often be applied to the system, rather than individual system features or services.

Non-functional requirements, as the name suggests, are the necessary conditions that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system characteristics such as reliability, response time, and warehouse occupancy.

Alternatively, they may define constraints on the computer system implementation such as the capabilities of I/O devices or the data statements used in interfaces with other systems. Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system. Non-functional requirements are frequently more critical than individual functional requirements.

System users can usually find methods to work around a system function that does not really meet their needs. However, failing to meet a non-functional requirement can imply that the whole system is unusable.

For example

- 1) Emails should be sent with a latency of no greater than 12 hours from such an activity.
- 2) The processing of each enquiry should be done within 20 seconds
- 3) The site should load in 4 seconds when the number of simultaneous users are greater than 11000

The implementation of these requirements can be diffused throughout the system.

There are two reasons for this:

1. Non-functional requirements may affect the overall structural design of a system rather than the individual components. For example, to ensure that the performance requirements are fulfilled, you may have to organize the system to reduce the communication between the components.
2. A single non-functional requirement, such as a security requirement, could produce several related functional requirements that define new system services that are necessary. In addition, it may also generate in accordance with the provisions that limit the existing requirements.

Non-functional requirements arise through user needs, due to the fact of budget constraints, organizational policies, the need for the interoperability with other software or hardware systems, or outside factors such as safety regulations or confidentiality legislation.

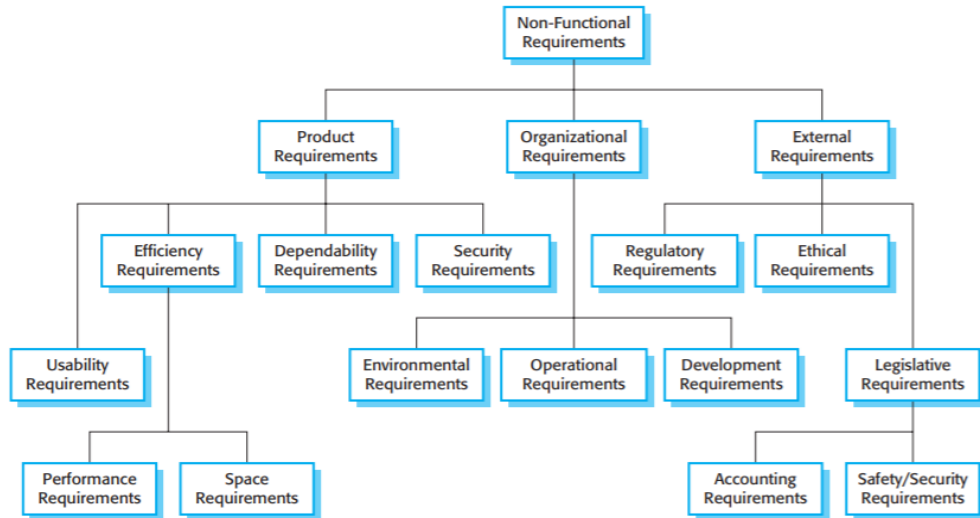


Fig: Classification of Non-functional Requirements

Non-functional requirements may come from the necessary characteristics of the software or from outside sources:

1. **Product requirements** These requirements stipulate or constrain the behaviour of the software.

Examples include performance conditions laid down on how fast the system must execute and how much memory it requires, reliability in accordance with the provisions that set out the acceptable failure rate, security requirements, and the usability requirements. The product prerequisite is an availability requirement that defines when the system should be available and the allowed down time each day. It says nothing about the features and clearly identifies a restriction that must be considered by the system architects.

2. **Organizational requirements** These requirements are comprehensive system requirements derived from policies and procedures in the customer's and programmer's organization.

Operational process specifications determine how the system can be used, implementation process requirements define the programming language, development environment, or process criteria that will be used, and environmental requirements define the system's operating environment. How

users authenticate themselves to the system is defined by the organisational requirement.

3. **External requirements** This broad heading encompasses all criteria arising from factors outside of the system and its creation process. This may include regulatory standards that specify what must be done in order for a regulator, such as a central bank, to allow the system for use; legal requirements that must be met to ensure that the system works within the law; and ethical requirements that ensure that the system is appropriate to its users and the general public.

A common problem with non-functional requirements is that customers or customers often propose these requirements as general goals, for example the ease of use, the ability of the system to recoup from failure, or rapid user response. The table below show the metrics that you can use to determine non-functional system properties.

Property	Measure
Speed	Processed deals/second User/event reply in the time Screen refresh time
Size	Bytes Number of ROM chips
Reliability	Mean time to failure. Probability of inaccessibility Rate of failure occurrence Availability
Robustness	Time to restart after failure. Percentage of events is caused by the failure. Probability of information from the corruption on failure
Portability	Percentage of target reliant statements Number of target systems
Ease of Use	Training period in which the Number of help frames

It is difficult, in practice, to separate functional and non-functional requirements throughout the requirements document. If the non-functional requirements are

specified separately from the functional requirements, the relationships between them may be hard to understand. However, you should explicitly emphasize the requirements that are clearly related to emergent system properties, such as performance or reliability. Non-functional requirements such as reliability, safety, and confidentiality requirements are also essential while defining the conditions necessary and should not be side-lined.

2.3 User Requirements:

The user requirements for a system should describe the functional and non-functional requirements so that they are easily understood by system users without thorough technical knowledge. They should only specify the external behavioural patterns of the system and should avoid, as far as possible, system design characteristics.

Consequently, if you are writing user requirements, you must not use the software jargon, structured notations, or formal notations, or explain the requirement by describing the system implementation. List of problems that occur when requirements are written in natural language.

1. **Lack of clarity** It is sometimes difficult to use language in an accurate and unambiguous manner without making the document wordy and hard to read.
2. **Requirement's confusion** Functional requirements, non-functional requirements, system objectives and design information might not be clearly distinguished.
3. **Requirement's amalgamation** Several different requirements can be expressed together as a single requirement.

User requirements that include too much data constrain the freedom of the system developer to provide innovative solutions to user problems and are difficult to understand. The user requirement should simply focus on the key and all the essential amenities to be provided.

To minimise misunderstandings when writing user requirements, following steps need to be taken:

- Invent a standard format and ensure that all requirement definitions follow that format.
- Standardising the format makes exceptions less likely and the necessary conditions easier to check. The format I use shows the initial requirement in

boldface, including a declaration of rationale with each user requirement and a reference to the more detailed system requirement specification.

- You may also include information on who proposed the requirement (the requirement source) so that you will know whom to consult if the requirement must be changed.
- Use language steadily.
- You should always distinguish between compulsory and desirable requirements. Mandatory requirements are requirements that the system must support and are typically written using 'shall'. Appropriate requirements are not essential and are written using 'should'.
- Use text highlighting (bold, italic or colour) to pick out crucial parts of the requirement.

Avoid, as far as possible, the use of computer terminology. Inevitably, however, the comprehensive technical terms will invade the user requirements.

2.4 System Requirements:

System requirements are expanded versions of the user needs that are used by software engineers as the starting point for the system design. They add details and explain how the user requirements should be offered by the system. They may be used as part of the contract for the application of the system and should therefore be a complete and consistent specification of the whole system. Ideally, the system requirements should simply explain the external behaviour of the system and its operational constraints. They should not be worried about how the system should be constructed or implemented. However, at the level of detail necessary to completely specify a complex software system, it is impossible, in practice, to exclude all information about the project.

Natural language is often used to write system specifications for the specifications as well as user requirements. However, because system requirements are more detailed than user requirements, natural language specifications may be confusing and hard to understand:

Natural language understanding depends on the specification readers and writers that uses the same words for the same notion. This leads to misunderstandings because of an ambiguity of natural language. A natural linguistic requirements specification is over flexible. You could tell you the same thing in completely different ways. It is up to the reader to find out when requirements are the same and when they are distinct. There is no easy way to modularise natural-language

requirements. It may be difficult to find all the related terms and conditions set forth. To discover the consequence of a transformation, you may need to look at every requirement rather than at simply a group of connected requirements.

Structured language Specification:

Structured natural language is a way of writing system specifications for the where the freedom of the requirements writer is limited, and all the demands are written in a standard way. The advantage of this approach is that it maintains most of the fluency and understandability of natural language but ensures that some degree of uniformity remains imposed on the specification.

Structured language notations restrict the terminology that can be used and use templates to specify system requirements. They can incorporate control constructs derived from programming languages and graphical emphasizing to partition the specification.

Insulin Pump/Control Software/SRS/3.3.2	
Function	Compute insulin dose: Safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1)
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered
Destination	Main control loop
Action: CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.	
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2
Side effects	None

Reference: Sommerville 8e Chapter 6 Software Requirements

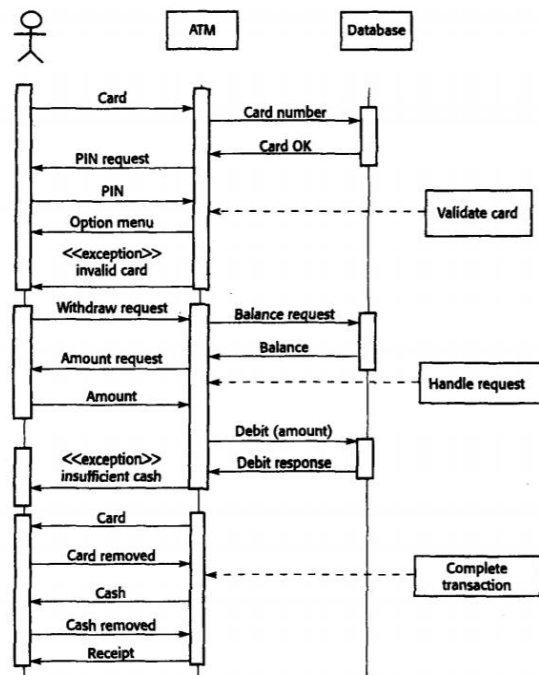
When a standard form can be used to specify the functional requirements, the following data should be included:

1. Description of the feature or entity being specified

2. Explanation of its inputs as well as where these come from
3. Description of its outputs and somewhere these go to
4. Suggestion of which other entities is used (the requires part)
5. Explanation of the action to be carried out
6. If a functional approach is used, a pre-condition setting out what should be true before the function is called and a post-condition if you specify what is true after the function is called
7. Description of the side effects (if any) of the operation

Graphical models are most useful when you need to show how state changes using sequence diagram there are three basic subsequence used:

1. Validate card the user s card is authenticated by checking the card number and user's PIN.
2. Handle request the user s request is handled through the system. For a withdrawal, the database must be queried to check the user's balance and to debit the sum withdrawn. Notice the exception here if the requestor does not have enough cash in their account.
3. Complete transaction the user s card will be returned and, when it is removed, the cash and receipt are delivered.



Reference: Sommerville 8e Chapter 6 Software Requirements

2.5 Interface Specification:

Almost all software systems must operate with existing systems which have already been implemented and installed in an environment. If a new system and the existing systems must work together, the interfaces of existing systems must be precisely specified. These specifications should be defined early in the process and incorporated into the requirements document.

There are three types of interfaces that may have to be defined:

1. **Procedural interfaces** where existing systems or sub-systems provide a range of services that are accessed by calling interface procedures. These interfaces have sometimes called Application Programming Interfaces.
2. **Data structures** that are passed from a particular sub-system to another. Graphical information from the models are the best entries for this type of description. If necessary, system descriptions in Java or python may be automatically generated from these descriptions.
3. **Representations of data** that have been established for an already existing sub-system. These interfaces are most common in the integrated, real-time system. Some programming languages like Ada support this level of specification. However, the best way to explain these is probably to use a diagram of the structure along with annotations explaining the function of every group of bits.

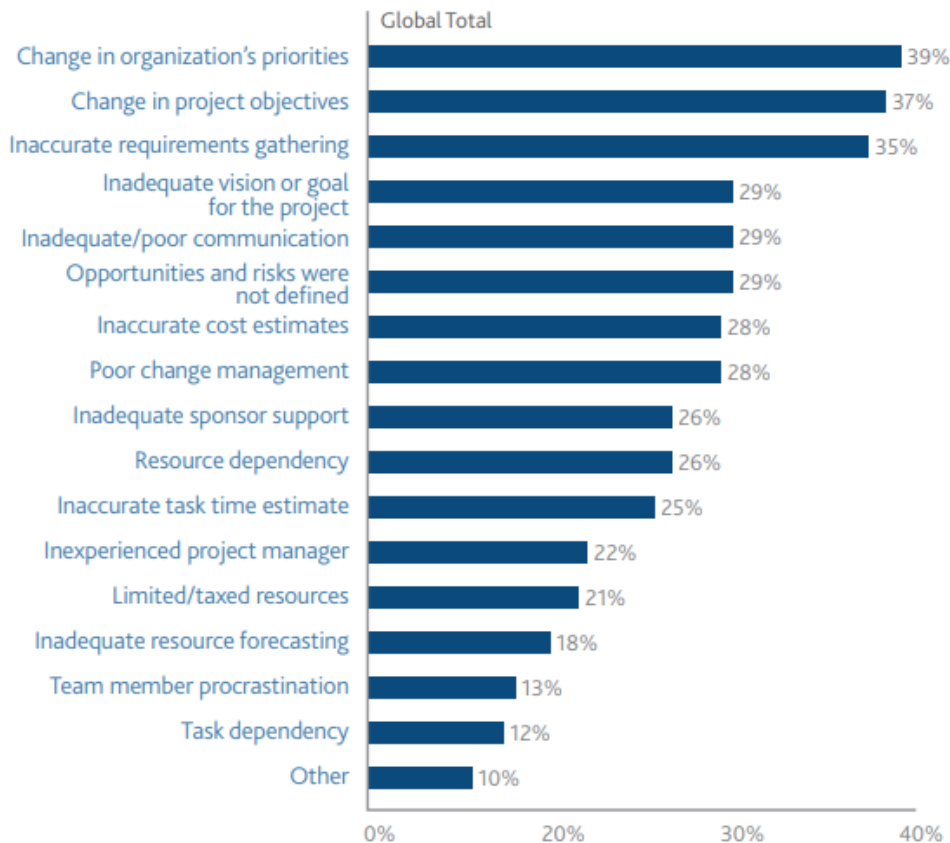
2.6 The Software Requirements Document

How can a client and provider reach a shared understanding on the concept of the product? It is not easy when they speak several languages. A customer defines a product at a high-level concept level, focusing on the external system behaviour: what it will be doing and how end-users will work with it. Meanwhile, developers think of the product is in terms of its own internal characteristics. That is why a Business Analyst steps in to convert a customer's needs into requirements, and further turn them into tasks for developers. This is initially done by writing **software requirements specifications**.

Poorly specified requirements inevitably lead to some functionality not being included in the application. Every assumption should be clearly communicated rather than just implied. For instance, NASA's Mars Climate Orbiter mission failed due to conflicting units of measure. Nobody specified beforehand that the mindset-control system and navigation software must both use the same metric or imperial units. For some, it went without saying, everyone else did not find it as obvious.

This is a widespread problem that keeps happening even to the best specialists if not prevented.

Q: Of the projects started in your organization in the past 12 months that were deemed failures, what were the primary causes of those failures? (Select up to 3)



Inaccurate requirements gathering is one of the top causes for project failure, Source: PMI's Pulse of the Profession

The manufacture of the requirements stage of a software development process is **Software Requirements Specifications (SRS)** (also called a **requirements document**). This report lays a foundation for software development activities and is constructing when whole requirements are elicited and analyzed. **SRS** is a formal report, which serves as a representation of software that enables the customers to review whether it (SRS) is in accordance with their requirements. Also, it includes user requirements for a system in addition to the detailed requirements of the system requirements.

The SRS is a specification for a particular software product, program, or a set of applications which perform functions in a specific environment. It serves several purposes depending on who is writing it. First, the SRS could be written by the client of a system.

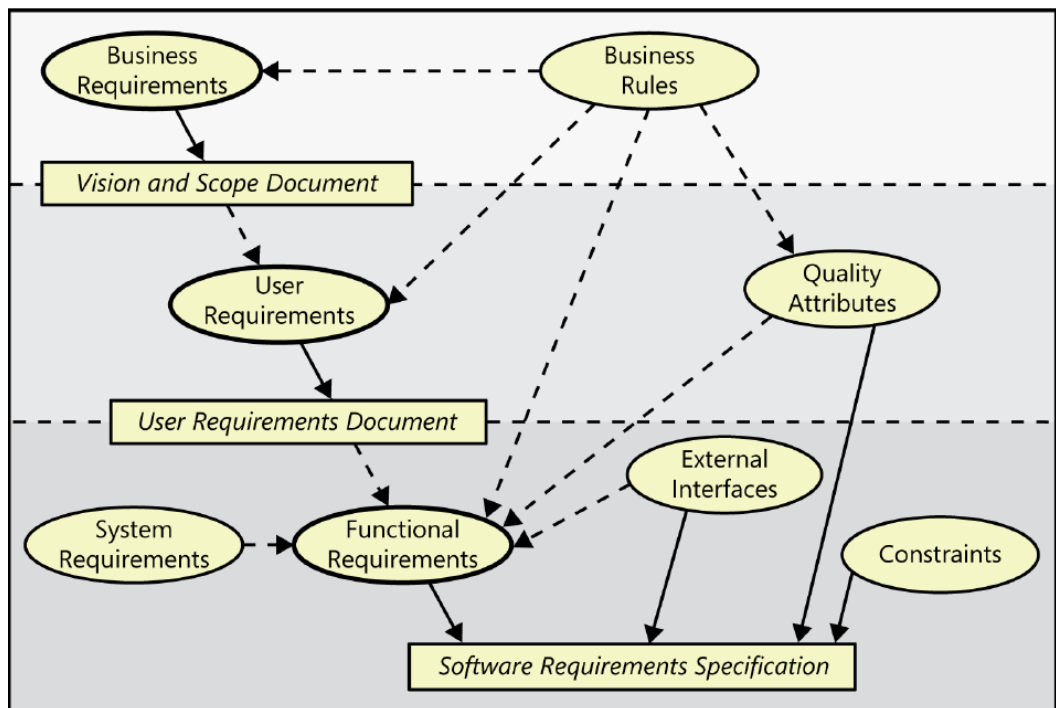
Second, the SRS could be written by a developer of the system. The two methods create entirely various situations and establish different purposes for the document altogether. The first case, SRS, is used to define the needs and expectation of the users. The second case, SRS, is written for different purposes and serves as a contract detail between customer and developer.

SRS is at the bottom of the software requirements pyramid, which goes like this:

Top tier – the high-level business requirements (BRs) dictating the aim behind the product,

Middle tier – user requirements (URs) that picture an end-user and their needs, and

Bottom tier – SRSs that specify the product's features are available in tech terms.



Solid arrows show how requirement types (ovals) are sorted into the documents (rectangles). While dotted arrows show which requirement type is the origin of or influences another one, Source: Software Requirements by Karl Wieggers Joy Beatty

SRS Software and Template

An SRS template provides a handy reminder of what kinds of knowledge to explore. Every software development organization would adopt one or more standard SRS templates for their projects. There are various templates available depending on the design class. Here is an example of an SRS template that works well for many types of projects.

- 1. Introduction**
 - 1.1 Purpose
 - 1.2 Document conventions
 - 1.3 Project scope
 - 1.4 References
- 2. Overall description**
 - 2.1 Product perspective
 - 2.2 User classes and characteristics
 - 2.3 Operating environment
 - 2.4 Design and implementation constraints
 - 2.5 Assumptions and dependencies
- 3. System features**
 - 3.x System feature X
 - 3.x.1 Description
 - 3.x.2 Functional requirements
- 4. Data requirements**
 - 4.1 Logical data model
 - 4.2 Data dictionary
 - 4.3 Reports
 - 4.4 Data acquisition, integrity, retention, and disposal
- 5. External interface requirements**
 - 5.1 User interfaces
 - 5.2 Software interfaces
 - 5.3 Hardware interfaces
 - 5.4 Communications interfaces
- 6. Quality attributes**
 - 6.1 Usability
 - 6.2 Performance
 - 6.3 Security
 - 6.4 Safety
 - 6.x [others]
- 7. Internationalization and localization requirements**
- 8. Other requirements**
- Appendix A: Glossary**
- Appendix B: Analysis models**

Source: *Software Requirements by Karl Wieggers Joy Beatty*

Structure with explanation of SRS:

Chapter	Description
Preface	This should define the anticipated readership of the document and describe its version history, including a rationale for the creation of a new edition and a summary of the changes made in each version.
Introduction	This should describe the necessity for the system. It should briefly describe its functions and explain how it will work with other systems. It should explain how the system fits into the overall business or strategic, to achieve the goals of the organisation commissioning the software
Glossary	This should define the technological terms used in the document You do not have to make any assumptions about the experience or expertise of the reader
User Requirement Definition	The service provided for the user as well as the non-functional system requirements should be outlined in this section. This description may use natural language, diagrams or other notations that are to be understood by customers. Product and process standards that should be followed should be specified.
System architecture	This chapter is required to submit an extremely high-level overview of the anticipated system structure showing the allocation of functions across system modules. Architectural components that are recycled must be highlighted.
System Requirements Specification	This should describe the functional and that are not related to-functional demands in more detail. If necessary, more detailed information can also be added to the non-functional

	requirements, e.g., interfaces with other systems may be defined.
System Models	This should set out a single result or multiple system models showing the relationships between the components in the system and the system and its environment. These might be object models, data-flow models, as well as the semantic data models.
System Evolution	This should describe the basic assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	These should provide detailed, particular information which is related to the application which is being developed. Examples of appendices that can be included are hardware and database descriptions. Hardware requirements define the minimal and optimal structures for the system. Database requirements define the logical organisation of the data that are used in the system and the relationships between data.
Index	several indexes to the document can be included. As well as a normal alphabetic index there might be an index of diagrams, an index of functions, etc.

2.7 Summary:

- Requirements for a software system set out what the current system should do and define the restrictions on its operation and implementation.
- Functional requirements are the declarations of the services that the system will have to provide or are descriptions of how certain calculations must be carried out.
- Non-functional requirements frequently constrain the entire system being developed and the development process being used. These could be product requirements, organizational requirements, or external requirements. They

often relate to the evolving properties of the system and therefore apply to the scheme.

- The software requirements document that is an agreed statement of the system requirements. It should be organized so that both approach customers and software developers will be able to use it.

2.8 References:

1. Beck, K. (1999). 'Embracing Change with Extreme Programming'. IEEE Computer, 32 (10), 70–8.
2. Crabtree, A. (2003). Designing Collaborative Systems: A Practical Guide to Ethnography. London: Springer-Verlag.
3. Kotonya, G. and Sommerville, I. (1998). Requirements Engineering: Processes and Techniques. Chichester, UK: John Wiley and Sons.
4. Larman, C. (2002). Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process. Englewood Cliff, NJ: Prentice Hall.

2.9 Questions:

1. Write a set of functional and non-functional specifications for the banking system, setting out its expected reliability and response time.
2. With an example write the customer requirements of ATM machine
3. Write a short note the significance of SRS.
4. List the advantages of SRS.

SOFTWARE PROCESSES

Unit Structure

- 3.0 Objectives
- 3.1 Introduction
- 3.2 Process and Project
 - 3.2.1 Project Management Benefits
 - 3.2.2 Project and Process Identification Benefits
 - 3.2.3 Difference between Project and Process
- 3.3 Component Software processes
 - 3.3.1 Software Requirement Specification
 - 3.3.2 Software Design and Implementation
 - 3.3.3 Software Testing
 - 3.3.4 Software Maintenance
- 3.4 Summary
- 3.5 Questionnaire
- 3.6 References

3.0 Objectives

After going through this chapter, you will be able to:

- define process and project
- understand how projects are classified
- state the benefits of project management
- state the different component software processes
- gain a brief understanding
- of each of the different component software processes

3.1 Introduction

A software process is a series of operations that lead to a software product being produced. These actions might include the development of a standard programming language like Java or C of the software from scratch. However, this is not necessary for commercial applications to emerge. New business software is typically currently produced by expanding and changing existing systems or by setting up and integrating system or off-the-shelf software.

Software processes are complicated, rely on humans to make decisions and judgements, much like other intellectual and creative processes. There is no perfect method and many companies have their own software development processes established. Processes have emerged to benefit from the capability and unique features of the systems built by the employees of a company. A more organised development approach is needed for certain systems, such as critical systems. A less formal, flexible approach is likely to be more successful in business systems with frequently changing needs

3.2 Process and Project

Project:

A project is a large action that entails a series of smaller activities that result in the creation of value in the form of a product or service. Customized planning, exploration, ideation, research, and talks about achieving stated goals are all part of it.

Example : - Building an eCommerce website for a customer .

The number of project stakeholders will vary depending on the size and scope of the project. If you're planning a large job, you'll need more personnel or time to do it.

What is a project?

Guided by a project manager and led by a project team, a project is a set of tasks needed to achieve a goal. It covers a scope, a set schedule, a project strategy and resources.

The management of projects is the discipline of project organization and execution. The project is expressed in the five-phase project life cycle:

Initiation of the project

The first step is the conception stage, which examines ideas, conducts research and makes decisions on possibilities. This decides the feasibility of the project.

Planning of the project

The second phase consists of considering the needs of stakeholders, setting out an objective, bringing together a project team and creating a project plan.

Running of the project

The project team is at this phase trying to develop results and fulfil the goals established in the project plan. Processes will be implemented; resources will be allocated and tasks will be allotted.

Monitoring of projects

The fourth phase of project management focuses on performance and monitoring of progress. Measures are taken to guarantee that everything is up-to-date and budgetary.

Closing of Project

The last stage is where all actions are stopped. The project must be closed, both successful and failing. Administrative activities are completed and evaluated to better future efforts. The evaluation will be carried out.

3.2.1 Project Management Benefits

Management of projects is a strong tool for organizations of all sizes which may provide numerous advantages.

It provides you with repetitive processes, rules, and strategies to assist you in the management of your projects. This can enable you to achieve tasks consistently, efficiently, on schedule and budget. It can help you achieve success.

List of benefits :-

- Improve your opportunities to achieve the desired result
- Get a new look at your project and how your company plan matches
- Priorities and make optimal utilization of your business resources
- Set the scope, time and budget from the beginning precisely
- Stay in time and maintain budget expenses and resources
- Enhance productivity and job quality
- Promote constant communication between employees, suppliers and customers
- Meeting the different demands of the project participants
- Mitigate project risk failure
- Enhance client satisfaction
- Acquire a competitive advantage and increase your profit

Classification of Projects

Projects are not classified in any standard way. These can, however, be divided into two major industrial and development categories in view of the project objectives.

Every one of them can be further classified taking into account the nature of the job (repetitive, non-repetitive), completion period (long term, short term etc.), cost, risk level (high, low, no risk), style of operation (wide, small, etc. ,construction, construction-operate-transfer etc.)

Industrial projects that are also referred to as commercial projects provide goods or services for the satisfaction of client demands and providing good returns to the investor/stakeholder.

These projects are further divided into two groups, namely demand-based and resource-based. The demand-led initiatives are meant to meet the latent demands of consumers and their complexity. Infrastructure for fertilisers, agricultural processing etc that are based on resources/provision use the existing resources, such as land, water, agriculture, raw materials and human resources.

Process:

A process is just a collection of activities that you have to do time and again. From cooking to paying electric power bills, everything you do in your daily life is connected to a process.

The procedure is broken down into stages. At least one stakeholder participates in each of these phases. The procedure starts, midway through and ends. For a single time, some processes are done. During the timeframe there are some routine processes that are repeatedly completed by the personnel.

For example, daily reporting to the management and approval for the employee's leave requests are distinct kinds of procedures.

A process has been created and repeated for internal business reasons. It involves a number of actions that have to be performed with each other to obtain a result.

Processes are an essential aspect of corporate knowledge and constitute many everyday activities. For example, the HR department has a procedure in place to hire new applicants and the development team has one to prioritise requests for features. A flux diagram is a popular way of viewing a process.

The aim of a process is to serve customer value business goals. They should be examined and upgraded periodically in order to ensure good business standards.

Types of processes :**Operational process:**

It focuses on the correct performance of a business/operational entity's tasks. In other words, "get things done" by staff. The customer service staff offers assistance to its customers is an example of this.

Management process:

It ensures proper performance of operational processes. This is where the management team ensures that work procedures are efficient and efficient. One example is the supervision of the tasks and operations of a project by a project manager.

Process of governance:

It guarantees that the enterprise/entity complies fully with applicable rules, standards and shareholder expectations.

3.2.2 Project and Process Identification Benefits**Project Identification advantages:**

Opportunity for cooperation - When you work on a project, it provides you an opportunity to work with new individuals.

Competence development - It enables you to grow your skills by trying to push your limitations.

Defined schedules - If you manage your project correctly, you will obtain a precise concept of how long you will achieve.

Advantages of identification of process

1. Bottlenecks -

You will discover new bottlenecks which were previously invisible when you routinely through repeated processes.

2. Smooth and organized flow -

It helps you develop a flow that can be effective in your job.

3. Control -

It allows you to manage your everyday chores, output standards or time taken during the entire procedure.

3.2.3. Difference Between Project and Process

The frequency of repetition is the difference between project and process. Projects are one-off, whereas processes are repeated.

The goals that are set are another other element. The objective of a project is to be successful. You want to reach there on time and on budget through the finish line. As projects are typically one-time tasks, a lot of planning needs to be done and the risks are sometimes enormous.

Project teams spend time ensuring that the project is delivered and that the risks are minimized.

In comparison, the focus is on optimization when conducting a process. There is little or no risk associated with processes (after all, it has become a standardized process for whatever reasons) and so typically the major aim is to refine it. The more you work, the simpler it is to adjust. Cost and time requirements can always be optimized.

The similarities between projects and process:

Both have duties to do. If a work is handed , one would want it to be done simply. It does not matter whether it is part of a project or process.

Projects and processes can also be found in each other.

Mini projects may involve processes.

For example, building a large branded shop, is more than a few times what happened, therefore any firm is supposed to follow a process.

However, a project that can be leasing a property in various places requires different processes and laws to be followed in establishing a store in a new place. It involves quality, time and costs, and the risk.

3.3. Component Software Processes

A software process is a collection of interconnected actions that results in the creation of a software product. These tasks might include creating software from the ground up in a standard programming language such as Java or C.

Business applications, on the other hand, are not often created in this manner. New business software is frequently created by expanding and altering existing systems or by configuring existing systems.

There are several software processes, however they all must involve four activities that are essential to software engineering:

1. **Specification of software:** The software's functioning and its limitations. It is necessary to define the operation.
2. **Software Design and implementation:** The software must adhere to the requirements. It is necessary to create.
3. **Testing (Verification of software):** To verify that the software performs what it says, it must be verified
4. **The evolution(maintenance) of software:** To satisfy changing consumer demands, the software must adapt.

These actions are present in all software processes in some way. They are, of course, complicated activities in and of themselves, including sub-activities like requirements validation, architectural design, unit testing, and so on.

Documentation and software configuration management are examples of supporting process activities. We generally talk about the activities in processes when we describe and discuss them. Specifying a data model, creating a user interface, and so on are examples of these activities and the order in which these operations are carried out.

Brief Description of the four activities involved in the software processes:

3.3.1 Software Requirement Specification

1. **Software Requirement Specification:**

The software requirements specification serves as the foundation for a contract between customers and contractors or suppliers about how the software product should work (in a market-driven project, these roles may be played by the marketing and development divisions).

The objective of software requirements definition is to reduce subsequent redesign by doing a thorough assessment of needs prior to the more specific system design stages. Software requirements specs should also offer a reasonable foundation for predicting product costs, risks, and schedules.

When used correctly, software requirements specifications may assist avoid software project failure.

The software requirements specification document contains adequate and required requirements for project development. The developer must have a clear and complete grasp of the products under development in order to derive the requirements.

Throughout the software development process, this is accomplished through comprehensive and ongoing communication with the project team and the client. The SRS may contain organizationally specific material or be one of the contract's deliverable data item descriptions.

A technical writer, a systems architect, or a software programmer often composes an SRS.

A SRS should contain sufficient information to complete the stated program. It not only outlines the program description in development but also the goal to which it is intended: what the software has to accomplish and how it should work.

An SRS documents typically includes these elements:

- * The purpose of developing the software
- * An entire description of Software
- * The functionality of the software or what the software should do in a production situation
- * Non-functional requirements :Outside interfaces or how the software interacts with hardware or other software.

A simple example is provided in the following sample of an SRS:

- i. Introduction
- ii. Purpose
- iii. Intended use
- iv. Scope
- v. Definitions
- vi. General description
- vii. User needs
- viii. Assumptions and dependence
- ix. System functional requirements
 - external interface
 - system requirements
 - system characteristics

In introducing your SRS, start explaining the objective of the product. You specify here the intended audience and how the product is to be used.

Here is how to design the aim: define the product scope , description of the value it will give ,show who will use the program in detail how the intended user will be helped with the software.

When you define the purpose of the product, outline how it works. Here you will provide an overview of the features of the programmer and how they suit the demands of the user.

You will also outline the assumptions you make regarding the functionality of the product and everything on which this depends in the existing technology environment.

Functional Requirements:

The aim of the new system you are building are functional requirements. You explain how the system works to aid with the tasks of a user. It defines how the system reacts to user entries and contains information on computations, data input and business operations. A thorough description of the software features and user demands may be considered for functional requirements. The system will not work without fulfilling the functional criteria.

Non-functional Requirements:

While functional needs indicate what a system does, non-functional needs explain how the system works. Non-functional needs do not influence the functioning of the system. They focus on user expectation and cover aspects such as performance, safety, trustworthiness, availability and usability, instead of focusing on user needs.

3.3.2. Software Design and Implementation:

Software Design:

The design functions and processes, including screen layouts, business rules, process diagrams, and other documentation, are covered in depth in systems design. This stage's result will be a collection of modules or subsystems that explain the new system.

The needs stated in the approved requirements document are used as the initial input in the design stage. As a consequence of interviews, workshops, and/or prototype efforts, a collection of one or more design components will be generated for each need.

Functional hierarchy diagrams, screen layout diagrams, tables of business rules, and business rules tables are examples of design components that define the intended system characteristics in depth.

Software Development:

The software solutions are built by teams depending on the design considerations made. By executing the solution, teams achieve the goals and outputs established during the software requirements collecting phase.

The development process may involve several individuals, new technologies, and unforeseen difficulties; nevertheless, development teams generally collaborate with tech leads, product managers, and project managers to unblock the process, make choices, and offer assistance.

Developers start creating the whole system using the programming language they have selected. Tasks are split in units or modules throughout the development stage and allocated to different developers.

Developers must follow specific established code directives at this stage. To create and implement code, they must employ programming tools such as compilers, interpreters and debuggers.

The process of coding involves transforming the system specifications into a code written in programming language which a computer can understand and perform the functions specified with the help of that code.

With efficient coding, one can greatly reduce the cost of testing and maintenance of the software at later stages. This leads to increased efficiency in the execution of the whole software project.

Now, coding does not only involve implementing the code in any appropriate language as the developer selects. Certain standards and standards, known as the coding standards, are set for this purpose.

In general, good companies creating software ensure that software engineers comply with these criteria to generate excellent software quality. Some organizations are rigorously following these guidelines, while some modify them to fit their needs and standards of quality.

Reasons for the coding standards:

- 1) Uniform appearance of the code --
Since so many coders and members are involved in the developing of the software, everyone follows the same standard, then there is a consistent look and naming conventions of the various modules which everyone can understand.
- 2) Easier understanding of the code:
Since everybody involved follows a common standard, it becomes easier for everyone to follow every piece of code. There are no ambiguities involved.
- 3) Lessening of dependency on a specific software developer

If a developer resigns in the middle of the project, and if another developer replaces him, it will become difficult for the new developer to comprehend the coding logic and make appropriate changes. Due to the common coding standards, the new hire can easily know the intricate logic and would be able to do the maintenance part.

4) **Encouragement of good programming practices:**

Good programming practices are given encouragement in the programmer's community which is something very commendable.

Once teams have packed and built their code, this step of work is completed.

3.3.3. Software Testing

Software testing is an important part of the development process. It includes a thorough examination of software to verify that it fits your client's needs and objectives.

The main objective of testing is to find all of the flaws and faults in the software before it is put into production. Software faults that are not resolved before deployment might have a negative impact on the client's business. The price of resolving those concerns would be prohibitive.

Testing, on the other hand, helps you to maintain software quality while also gaining your clients' trust and confidence. Furthermore, because the finished product will work precisely, consistently, and dependably, it will require less maintenance.

Requirement Testing , Unit testing, integration testing, system testing, and acceptance testing are the stages of software testing in total. With that in mind, these four phases may be divided into two categories, the first two of which are verification stages and the final two of which are validation stages.

The main distinction between the two is that verification testing is based on the methods that were utilized throughout development. The validation step, on the other hand, examines the final product's functioning and, in the end, incorporates user feedback.

a. Requirements Testing

It is important to check that you can not only build test cases against every need, but the definitions are clear and unambiguous in their testing of your requirements, whether in business, functional or technical. Make it incorrect and you may suffer extra rework expenses, lost deadlines, probable system failures and harm to the reputation of your organization.

b. Unit Testing

A unit refers to a small software segment which the developers have written and maintained. Testing exercises a small portion of the feature and controls the outcomes.

One of the greatest advantages of automated testing is that it can be done when a code part is modified and that problems may be fixed as soon as feasible. Before software developers give software to test testers for extra testing, it is fairly frequent.

c. System Testing

System tests are the stage of software tests where separate software components are integrated and fully tested. This is particularly essential since it validates that the functional areas perform according to expectations and that each system works together. Regardless of whether every unit is being tested, the software may not function as necessary if you don't verify it is appropriately integrated.

d. Regression Testing:

Regression testing checks if the previously created and operating program still functions appropriately when it is modified. Changes might involve improving software, fixing bugs or updating infrastructure.

You can ensure that fresh modifications to an application have not produced regression problems by restarting test scenarios and causing parts that were originally working.

e. Acceptance Testing

Acceptance tests (or user acceptance tests – UAT) are carried out to verify if the system is ready. UAT is going to strive to check operational and business needs. Requirements might be misunderstood and executed occasionally in a way that does not satisfy user or business goals.

3.3.4. Software Maintenance

The process of changing a software product after it has been delivered to the client is known as software maintenance. Software maintenance is the process of modifying and updating software programs after they have been delivered in order to fix bugs and enhance performance.

In general, software remains active for a long time after it is first installed, and it needs ongoing maintenance to guarantee that it continues to run at peak levels.

Software programmers routinely provide software patches during the maintenance phase of the software life cycle to address changes in an organization's needs, to rectify issues pertaining to faults in the software, or to tackle possible security risks.

During the maintenance phase, designers resolve any issues that are identified in order to ensure that the software performs as intended or to add new features to the product.

Software Maintenance is Required to:

- Correct Errors.
- Enhance the design.
- Enhance the situation.

- Connect to other systems.
- Allow various hardware, software, system features, and telecommunications facilities to be used with different programs
- Legacy software should be migrated.
- Software should be retired.

Depending on the nature of the program, the sort of maintenance required may change over time.

It might be a simple routine maintenance operation, such as fixing a problem found by a user, or it could be a big event in and of itself, depending on the scale or nature of the maintenance.

According to their characteristics, the following are some forms of maintenance:

1. Corrective Maintenance -

This comprises adjustments and updates made to repair or fix problems that are either discovered by users or concluded by user error reports.

2. Adaptive Maintenance -

It entails making changes and updates to the software product to keep it current and tailored to the ever-changing world of technology and business.

3. Perfective Maintenance -

It entails making changes and upgrades to the program to maintain it useful for a long time. It offers new features as well as additional user requirements in order to improve the software's dependability and performance.

4. Preventive Maintenance -

It entails making changes and updates to the program to avoid future issues. Its goal is to address issues that aren't as serious right now but might become problematic in the future.

Real-world issues impacting maintenance costs

- Older software's, which were designed to run on sluggish computers with limited memory and storage, are unable to compete with freshly released improved software on contemporary technology.
- Maintaining outdated software gets more expensive as technology progresses.
- Most maintenance engineers are inexperienced and rely on trial and error to solve problems.
- Frequently, modifications can easily harm the software's original structure, making further changes difficult.
- Changes are frequently left unrecorded, which may lead to future problems.

The software maintenance activity is the phase that lasts the longest compared to other activities. This is because any software has a life for a specific length of time.

This is because the software keeps on getting used until it becomes totally outdated or the business process undergoes a drastic change requiring completely new software or automation strategies.

3.4 Summary

A project is a huge venture that entails a sequence of smaller activities that lead to creation of value in the form of a product or service. Appropriate planning, study, idea formation, research, and brainstorming sessions about achieving stated goals are all part of it.

Management of projects is a strong way for companies of all sizes which may provide a lot of advantages.

Projects are not grouped in any standard way. These can, however, be divided into two sizable industrial and development categories keeping in view the project objectives.

Every project can be further classified thinking about the kind of the job (repetitive, non-repetitive), length of time taken (long term, short term etc.), cost, risk level (high, low, no risk), nature of operation (wide, small, etc. ,construction, construction-operate-transfer etc.)

A process is an activity that you have to do time and again. From making tea to paying internet bills, everything you do in your daily life is a process.

The number of repetitions is the difference between project and process. Projects are one-off, whereas processes are repeated.

Projects and processes can also be found in each other.

Components of software processes:

- 1. Specification of software:** The software's functioning and its restrictions. It is necessary to define the operation.
The software requirements specification serves as the baseline for a contract between customers and contractors.
- 2. Software Design and implementation:** The software must stick to the requirements. It is necessary to create. Developers start building the entire system using the programming language they have selected.
- 3. Testing (Verification of software):** To verify that the software performs what it states, it must be verified. It includes a detailed examination of software to verify that it fits your client's needs and aims.
- 4. The evolution(maintenance) of software:** To satisfy changing consumer demands, the software must adapt. In general, software remains active for a long time after it is first installed, and it needs ongoing maintenance to guarantee that it continues to run at peak levels.

3.5 Questions

- 1) What is a project?
 - 2) What is a process?
 - 3) What is the difference between project and process?
 - 4) What are the different types of processes?
 - 5) What are the advantages of project management?
 - 6) What are the steps in project management?
 - 7) What are the different software component processes?
 - 8) Explain Software Requirement Specification Stage.
 - 9) Explain Software Testing phase.
 - 10) Write in detail about the Software Maintenance phase.
 - 11) What is the difference between verification and validation?
-

3.6 References

<https://www.manage.gov.in/studymaterial/PM.pdf>

Software Engineering Edition, Ian Somerville, Pearson Education

Software Engineering, Pankaj Jalote, Narosa Publication

Software Engineering, A practitioner's approach, Roger Pressman, Tata McGraw Hill

<https://www.projectcentral.com/blog/project-management-benefits/>

<https://appian.com/bpm/what-is-process-improvement-in-organizational-development-.html>

<https://www.manifera.com/6-basic-steps-software-development-process/>

<https://www.ibm.com/topics/software-testing>

<https://www.plutora.com/blog/verification-vs-validation>

SOFTWARE DEVELOPMENT PROCESS MODELS

Unit Structure

- 4.0 Objectives
- 4.1 Introduction
- 4.2 Waterfall Model
 - 4.2.1 Merits and Demerits
- 4.3 Prototyping Model
 - 4.3.1 Phases
 - 4.3.2 Types of prototype models
 - 4.3.3 Merits/Demerits
- 4.4 Iterative Development Model
 - 4.4.1 Where it is suitable
- 4.5 Rationalized Unified Process Model
 - 4.5.1 Phases of RUP Model
- 4.6 RAD Model
 - 4.6.1 RAD Phases
 - 4.6.2 Benefits/Drawbacks
- 4.7 Timeboxing Model
 - 4.7.1 Pros and Cons
- 4.8 Summary
- 4.9 Questionnaire
- 5.0 References

4.0 Objectives

After going through this chapter, you will be able to:

- state different software development process models
- understand waterfall model and its use
- understand prototyping model, its benefits , its use
- understand iterative development model
- gain a brief understanding of rationalized unified processes
- gain a brief understanding of RAD and Timeboxing Model

4.1 Introduction

A model software process is a reduced software process representation. Each model describes a process from a specific point of view and therefore only gives partial information. We see the process structure but not the particular activity information.

These generic models are not software process descriptions. Instead, they are abstractions that may be used to describe software development techniques.

These models do not exclude one another and are frequently employed jointly, particularly in the creation of complex systems. For big systems, some of the finest characteristics of the waterfall and progressive models have to be combined. To build a software architecture, you need to be aware of the fundamental system needs to support these requirements. You can't gradually build this. Sub-systems can be built using several techniques inside a larger system.

Parts can be specified and created by means of a waterfall-based approach which are fully known. Parts of the systems, such as the user interface, which are difficult to describe beforehand, should be built utilising a gradual approach always.

4.2 Waterfall Model

The first process model to be introduced was the Waterfall Model. The understanding and use is extremely easy. The Waterfall model is the oldest SDLC software development technique.

The waterfall model demonstrates a linear sequential flow of the software development process. This means that every stage of the development process starts only when the preceding stage is complete. The stages do not overlap in this waterfall model.

The different sequential phases of the Waterfall model are -

1) Feasibility Study:

The major objective of this phase is to assess whether the software can be developed financially and technically.

This feasibility study includes a study of the problem and then feasible solutions for solving the problem are then determined. Based on their advantages and disadvantages, these alternative solutions are assessed, the best option is picked and the following phases of this model are implemented.

2) Analytical and specification requirements:

The objective of the requirement analysis and specification stage is to accurately comprehend and document the customer's demands. Two distinct activities constitute this phase.

- a) Collection and analysis of requirements: The customers first collect all software needs and then assess the collected requirements. The objective of the analysis stage is to eliminate incompleteness and inconsistencies.
- b) Specification of requirements: These evaluated requirements are documented in an SRS document. SRS document is a contract between customers and the development team. An examination of the SRS document can resolve any future disagreement between customers and developers.

3) Design:

The objective is to translate into a structure appropriate for implementation in a certain programming language the criteria described in the SRS document.

4) Coding and testing of units:

in the design of software coding phase, any appropriate programming language is used to build source code. Every module designed is therefore programmed. The objective of the test phase is to see whether or not each module works properly.

5) Integration and testing:

Integration of several modules will be carried out shortly after coding and testing of every unit. Incremental integration of different modules takes place over several steps. The intended modules are introduced to the partially integrated system during each integration phase and the resulting system tested. Finally, the full work system is realized and system testing is carried out after all modules have been successfully integrated and tested.

6) Maintenance Phase :

The software deployed is upgraded and new features are added if required. Any bugs discovered later on are taken care of.

4.2.1 Waterfall Model Merits and Demerits**Waterfall Model Benefits**

- It's easy to grasp and use.
- Due to the rigidity of the model, it is easy to control - each step has specified results and a method for revision.

- In this model phases are completed one at a time. They do not overlap.
- For smaller projects, the waterfall model works effectively, when needs are well stated and well-understood.

Demerits of the model waterfall

- Once the application is in the testing phase, anything that was not properly thought-out at the ideation stage is tough to go back and modify.
- Until late in the life cycle, no functioning software is generated.
- High danger and insecurity levels.
- Not suitable for object-oriented projects.
- Not suitable for projects where requirements are ever changing.

When Waterfall Model is used :

- Only when the needs are fully understood, unambiguous and fixed is this paradigm employed.
- The definition of the product is stable.
- The requirements are very clear/unambiguous
- Extensive resources with the necessary knowledge are readily accessible
- Project is short term.

4.3 Prototyping Model

Prototyping model is a software development approach that constructs, tests and revises the prototype up to an acceptable level. It also provides a basis for producing the final software system. It works well in situations in which the needs of the project are not fully known. It is an iterative process between the developer and the customer and it is based on trial and error.

4.3.1. Phases

Steps in Prototyping model

Step 1: Collection and analysis of requirements

The need analysis begins with a prototype model. The system requirements are established in depth at this phase. The system users are interviewed throughout this phase to find out what their expectations are about the system.

Step 2: Quick Design

The second stage consists of a preliminary design. A basic system design is built at this stage. It's not a full design, however. It offers the user a brief picture of the system. This design contributes to the prototype development.

Step 3: Prototype construction

A real prototype is created in this step based on the information collected from the design in the previous design. The resulting system is a tiny functioning model.

Step 4: First user assessment

At this step, the system suggested for an initial review is presented to the customer. The strength and weakness of the working model can be identified. The consumer gives feedback and suggestions to the software coders.

Step 5: Prototype refinement

Should the user not be satisfied with the present prototype, the prototype must be modified according to user feedback.

Step 6: Product implementation and maintenance:

The system is fully tested and implemented in production after the final system is built based on the final prototype. Regular maintenance is carried out to minimize downtime and avoid large-scale breakdowns.

4.3.2 Types Of Prototype Models

Types of Prototyping models

1) Quick Throwaway

It is rapidly designed to display the visual appearance of the requirement. The input of the client helps to alter the needs and the prototype is built again till the need is based. A prototype created is rejected and does not form part of the prototype that was eventually approved. This method helps to explore ideas and to receive immediate feedback on consumer needs.

2) Evolutionary

Here the prototype is progressively improved based on the feedback of the customer until it is accepted at last. It can save you time and effort. Because building a prototype from scratch may often be quite difficult for every interaction of the process. This approach is useful for a project that employs a new, poorly understood technology.

3) Incremental

The final product is divided and developed separately into numerous tiny prototypes. The many prototypes will eventually be fused into a single product. This approach helps to shorten the period between input between the user and the development team.

4) Extreme:

It is mainly used for web development. There are three successive stages.

HTML format is included as the basic prototype and a services layer to mimic data processes.

The services are implemented in the final prototype and integrated.

4.3.3 Benefits and Demerits**Benefits :**

- Development involves users actively. In the earliest stage of the software development process, problems can be discovered.
- Identification of features that decrease the chance of failure, as prototyping is also a risk reduction activity.
- Assists team members with efficient communication
- Satisfaction with the consumer occurs because the product may be realized very early.
- Software rejection is scarcely possible.

Demerits :

- It is a slow process.
- The development costs for a prototype are a complete waste, as the prototype is finally discarded.
- Excessive demands for change may be encouraged through prototyping.
- Sometimes consumers may be not prepared to engage for the extended duration of the iteration cycle.
- When the prototype is reviewed by the client, there may be too many differences in the software requirements.

4.4 Iterative Development Model

Iterative development is a software development method that divides the development process into smaller pieces of a major program. Each element, known as the "iteration," reflects the whole process and comprises phases for planning, design, development and testing.

Iterative Model Advantage:

- Early and fast in the life cycle, certain working functions can be created.
- Early and periodic results are achieved.
- One can plan development parallelly.

- Growth in development may be assessed.
- It is less expensive to modify the scope/needs.
- Smaller iterations make it simpler to test and troubleshoot.
- Risks are recognized and resolved and each iteration is a milestone easily controlled.
- Risk management is easier.
- The operating product is supplied with each increase.

Iterative Model Disadvantages:

- It is not appropriate for minor tasks.
- It may require additional resources.
- Due to imprecise criteria, design might be altered over and over again.
- Changes in budget requirements may cause.
- The completion date of the project was not certain as the specifications had changed.

4.4.1 Where it is Suitable

In the following circumstances, this model is most commonly employed :

Clearly specified and understood are the requirements for the entire system.

Important needs must be stated, although certain features or upgrades sought may be developed over time.

The development team is using and learning a new technology during the project's work.

No resources with the necessary skills are available and are scheduled for usage in certain iterations on a contractual basis.

In the future, there are some risky features and objectives of the software project may change.

4.5 Rationalised Unified Process Model

The **Rational Unified Process (RUP)** is an iterative software development process framework created by the Rational Software Corporation, IBM since 2003.

The RUP is built on some building elements that describe what is to be developed, what skills are needed and how certain development goals are to be reached step by step. The following are the primary building elements:

- a) Roles (who) – The role specifies a collection of linked talents, powers and tasks.

- b) Work products - a work product, comprising all papers and patterns created during the process, symbolizes anything that is a result of a task.
- c) Tasks (How) – A task represents a work unit allocated to an important role.

The RUP has established a four-phase project life-cycle. These phases permit a high-level presentation of the process, comparable to what a model like the 'waterfall' might do, however the key to the process resides essentially in the development iterations in all the phases. Each phase also includes a major aim and milestone, which marks the completed goal.

4.5.1 Phases

A) Inception Phase

The main purpose is to make sufficient use of the system to validate initial costs and budgets. In this step, a business case is created which comprises the company environment, success parameters(anticipated revenue, market recognition, etc.). A basic case model for the usage, project plan, first risk assessment and project description are created to support the business case (core project needs, requirements and key features).

The project is examined according to the following criteria:

- a. Stakeholder competition in defining scope and estimating costs / timing.
- b. Understanding of requirements as shown by primary use case.
- c. Cost/plan estimations, priorities, risks and development process credibility
- d. The depth and width of any established architectural prototype.
- e. Creation of a basis for comparing current expenditure to anticipated expenditure.

If this phase is not passed by the project, it can either be repeated or terminated to better fulfil the requirements.

B) Elaboration Phase

The main aim is to reduce the main risk elements discovered via analysis up to the end of this phase. This phase is the beginning of the project. In this phase, problem domain analysis is carried out and the project architecture takes its fundamental form.

The results of this phase are:

- 1. A use case model that identifies usage cases and actors, and most of the usage case descriptions are developed. It should be 80% complete.
- 2. A description of the software architecture in the creation of software systems.

3. An architecture which executes scenarios of important architectural use.
4. Business Case and list of risks.

C) Construction Phase

The main aim is the construction of the software system. At this stage, the major focus is on component development and other system features. This is the stage in which the bulk of coding occurs. In larger projects, many iterations might be produced in order to break the uses into manageable pieces to build prototypes.

D) Transition Phase

The main goal is to 'transition' the system from development to production and make it possible for the end user to use and understand it. This phase involves educating end users and maintenance staff and beta testing the system to validate it against the expectations of end users. The system also undergoes assessment phases, in which any developer that doesn't do the needed work is replaced. The product is also tested against the degree of quality specified at the startup stage.

When all targets are accomplished, the milestone for release is achieved and the development cycle is completed.

RUP is built on six best practices, which have developed into the six fundamental principles that helped thousands of customers supporting software development projects worldwide, as well as inside IBM itself.

These are the essential principles:

- ❖ Process adjustment.
- ❖ Balance Competitive priorities of stakeholders.
- ❖ Collaborate across teams.
- ❖ Provide value on a substantial basis.
- ❖ Increase the abstraction level.
- ❖ Concentrate on quality continuously.

The following are the best practices:

- ❖ Iteratively develop.
- ❖ Requirements management.
- ❖ Use Architecture Component.
- ❖ Use a visual model.
- ❖ Verify quality continuously.
- ❖ Change management.

4.6 Rapid Application Development Model

RAD is a linear sequential paradigm of software development. If the requirements are clearly understood and defined and the scope of the project is a constraint, the RAD method enables a development team to produce a completely working system in a short amount of time.

RAD is an approach that allows products to be built more quickly and quality through:

- i. Collecting needs through workshops or focus groups
- ii. Prototyping and early repetitive user testing
- iii. Software component recycling
- iv. A strict schedule referring to advances in design
- v. Less formal reviews

4.6.1 Rad Phases

The various phases of RAD are:

1. Business modelling: The flow of information across business operations is defined by answering questions like which data is driven, which data is created, who is generating them, where the information is going, who is processing it and so forth.
2. Modeling of data: The information gathered through business modelling is refined into a collection of data objects (entities) necessary to sustain a company. Each entity's characteristics are recognized, and their relationship is established.
3. Process Modeling: In the data modelling step, the data object defined is turned into the required data flux for a business function. Descriptions of processing are developed to add, alter, delete, or recover a data item.
4. Generation of application: automated software building technologies are utilized; even 4th GL methods are employed.
5. Testing and turnover: Since RAD emphasizes reuse of some components of programming that have previously been tested, it leads to decrease in the testing time overall. But it is necessary to test the new part and to complete all interfaces.

4.6.2 Rad Model Benefits/Drawbacks

RAD Model Benefits

- i. It's flexible to alter this model.
- ii. Changes can be adopted in this model.
- iv. Every phase of RAD gives the consumer the utmost importance.
- iv. The development time has been decreased.
- v. The reusability of features is increased.

The RAD Model Drawbacks

- i. Highly talented designers were required.
- ii. Every app is not RAD compatible.
- iii. We can't utilize the RAD approach for smaller projects.
- iv. It is not appropriate for the high technical risk.
- vi. User participation required.

When to use the RAD Model?

When the system has to generate the modelling project in a brief period of time (2-3 months).

When it is known about the requirements.

If technical risk is limited.

If a system has to be built that modularized in 2-3 months.

Only if the budget permits automatic code generation tools. It should be employed.

4.7 Timeboxing Model

The time boxing process is the greatest means of increasing productivity and dividing tasks into assigned time periods. This time management approach gives the possibility of limited time spent in advance on a certain task. The ultimate objective is to identify and limit the time spent on a given activity.

Timeboxing is the treatment to prevent time limitations from being exceeded. This approach guarantees a timely delivery and provides progressive insight through the allocation of maximum time units. These times are also known as "periods," "timeboxes" or "items."

For timeboxing the whole project term is often split into many smaller segments. This produces mini-projects that become part of the whole project. The project may be readily adjusted by assessing at the end of a time period.

Steps:

Step one: Assess the length of time each sub-activity needs.

Step two: Give every sub activity a certain length of time; this is a time box.

Step three: Please include breaks in the time boxes schedule.

Step four: Also take unanticipated events, like breaks and unexpected visitors, into account in all time boxes.

Step five: Use a time-set to watch individual time-boxes closely.

Step six: Assess each check-box; if the sub-activity is halted or does not fit in a time-box, the reason(s) have to be taken into account.

4.7.1 . Timeboxing Model Pros and Cons

Timeboxing advantages

There are numerous advantages to using timeboxing. It will not just meet deadlines, but also allow staff to focus more effectively, focusing on their work. It avoids discontinuation because time constraints force you to ignore distractions and give priority to your job.

It also guarantees that the perfectionist, who takes a long time to accomplish an activity, gets 'completed' inside the time frame.

In particular, time-boxing can help, as just one action needs to be performed every day.

Timeboxing is also used for project management. In a variety of times, projects are planned, each with its own target, budget, and deadline. The so-called 'critical path,' the time period with less freedom, is evident if you know the least time to complete a project in advance.

It not only offers a definite period of time, but also guarantees that the budget is not overcome. The decline of smaller activities and the recruitment of more employees at peak times may be considered in order to fulfil deadlines.

In the time boxing paradigm, like in the iterative enhancement approach, development takes place iteratively.

However, each iteration in a time box model takes place within a set length timebox.

The designed functionality is modified to match the timebox duration. In addition, each timebox is separated into a predetermined stage sequence in which each stage carries out a clearly defined job, which may be done separately (analysis, implementation and deployment).

This approach also needs about equal time for each phase so that the pipeline idea reduces the time for development and releases of products. For every step there is a specialized staff so that the job is possible.

Disadvantages:

Project management becomes complex

It is not suitable for projects where entire duration cannot be divided into multiple iterations of equal duration.

4.8 Summary:

- The waterfall model demonstrates a linear sequential flow of the software development process. This means that every stage of the development process starts only when the preceding stage is complete. The stages do not overlap in this waterfall model.
- Prototyping model is a software development approach that constructs, tests and revises the prototype up to an acceptable level. It also provides a basis for producing the final software system. It works well in situations in which the needs of the project are not fully known. It is an iterative process between the developer and the customer and it is based on trial and error.
- Timeboxing is the treatment to prevent time limitations from being exceeded. This approach guarantees a timely delivery and provides progressive insight through the allocation of maximum time units. These times are also known as "periods," "timeboxes" or "items."
- RAD is a linear sequential paradigm of software development. If the requirements are clearly understood and defined and the scope of the project is a constraint, the RAD method enables a development team to produce a completely working system in a short amount of time.
- The RUP is built on some building elements that describe what is to be developed, what skills are needed and how certain development goals are to be reached step by step. The following are the primary building elements.
 - a) Roles (who) – The role specifies a collection of linked talents, powers and tasks.
 - b) Work products - a work product, comprising all papers and patterns created during the process, symbolises anything that is a result of a task.
 - c) Tasks (How) – A task represents a work unit allocated to an important role.
- Iterative development is a software development method that divides the development process into smaller pieces of a major software.

4.9 Questions

- 1) Explain Waterfall model. List its advantages and disadvantages.
- 2) What is the RAD model?
- 3) Explain different types of prototyping models.
- 4) What is meant by a prototype software model?
- 5) Explain in brief iterative development model.
- 6) Explain the time boxing model.

- 7) Write a note on RUP.
- 8) What are the different phases in the Waterfall Software Model?
- 9) List the merits and demerits of each software development model.
- 10) Explain any two phases of the Rationalised Unified Process Model.
- 11) When is the waterfall model used and when is the iterative development model used?
- 12) What are the principles and practices in the RUP model?

5.0 References

<https://www.manage.gov.in/studymaterial/PM.pdf>

Software Engineering Edition, Ian Somerville, Pearson Education

Software Engineering, Pankaj Jalote, Narosa Publication

Software Engineering, A practitioner's approach, Roger Pressman, Tata McGraw Hill

<https://www.scnsoft.com/blog/software-development-models>

<https://www.synopsys.com/blogs/software-security/top-4-software-development-methodologies/>

www.wikipedia.org

AGILE SOFTWARE DEVELOPMENT

Unit Structure

- 5.0 Objectives
- 5.1 Introduction
- 5.2 Agile Methods
 - 5.2.1 Agile Principles
 - 5.2.2 Agile Method Limitations
- 5.3 Plan Driven and Agile Development
 - 5.3.1 Issues to consider when balancing agile and plan based development
- 5.4 Extreme Programming
 - 5.4.1 Testing in XP
 - 5.4.2 Pair Programming
- 5.5 Agile Project Management
 - 5.5.1 Stages in Scrum
- 5.6 Scaling Agile Methods
 - 5.6.1 Challenges in implementation
- 5.7 Summary
- 5.8 Questionnaire
- 5.9 References

5.0 Objectives

After going through this chapter, you will be able to:

- Know about the agile methods
- Understand plan driven and agile development models
- Know about extreme programming
- Understand agile
- Project management
- Gain a brief understanding of issues of scaling agile methods

5.1 Introduction

Corporations currently operate in a fast-changing global context. They must adapt to the introduction of competitive products and services and to the changing economic conditions. Software is part of most company activities; such that new software is rapidly produced to seize new possibilities and react to competitive pressure.

So, the most important need for software systems is now, consequently, fast development and delivery. In reality, many companies are prepared to compromise the quality of software and the requirements in order to obtain a quicker deployment of the required software.

Since these companies operate in a dynamic environment, a comprehensive set of solid software requirements are frequently nearly difficult to draw out. The original requirements alter unavoidably since consumers cannot foresee how a system affects their work habits, how it interacts with other systems and which user actions should be automated.

It is only possible that the actual needs become obvious once a system is supplied and users grow familiar with it. Still though, because of external causes the needs will probably alter fast and unpredictably.

When supplied, the software may be out of date. Software development procedures that are designed to fully set the requirements and design, create and test the system will not be designed for fast developments in software. The systems design or implementation should be updated and tested when needs change or as requirement issues are found.

Consequently, a typical waterfall or a specification process is usually extended, and once it has first been specified, the completed program is provided to the client. A plan-driven approach is the correct one for some types of software, such as safety-critical control systems, where full system analysis is needed. But this might pose serious issues in a rapidly changing corporate environment.

The initial purpose for its procurement may have so dramatically altered when the program is available for use that the software will be worthless. Development procedures focused on quick software development and delivery are therefore crucial for commercial systems.

Rapid procedures in software development are designed for speedy software production. The program has not been built as a single unit but as a number of increases, each with additional system capability.

5.2 Agile Methods

There was a broad belief in the 1980s and early 1990s that careful project planning, formal quality assurance, the use of Case tools backed analysis and design methodologies and regulated, and rigorous software development processes were the best way to produce better Software.

This idea was taken from the community of software engineering, which developed huge, long-standing software systems such as aerospace and governmental systems. This software has been created for several organizations by huge teams.

Teams were typically scattered geographically and worked on the software for a long time.

The planning, development and documentation of the system constitute a considerable overall task. This overall cost is justified in coordinating the activities of numerous development teams if the system is a crucial system and many individuals take part in the maintenance of software throughout their lives. However, when this heavy-weight, plan-driven method is applied to SMEs, the overhead is so high that it dominates the software development process.

It takes more time to create the system than to design and test the software. As the needs of the system change, rework is required and the design and design of the program must, in theory, at least alter. In the 1990s, several software designers proposed new 'agile methodologies' due to their dissatisfaction with this high-profile approach in software engineering. The development team therefore focused not on the design of the software itself.

Agile techniques often rely on a progressive approach to the design, development, and delivery of software. They are ideally suited to the creation of an application when the system must change quickly in the process of development. The aim is to offer work software to clients rapidly, who may afterwards submit new and modified needs to be integrated in the system subsequently. They want to eliminate bureaucracy by avoiding long-term work with doubtful value and by removing documents which are unlikely to be needed.

In the agile Manifesto accepted by many of the major creators of these techniques, the concept underlying agile methods is expressed.

This manifesto states:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan.
- That is, while there is value in the items on the right, we value the items on the left more.

5.2.1 Agile Principles

Although all of these agile techniques are founded on the concept of progressive development and execution, they suggest various procedures to do so. But, according to the agile manifesto, they share a set of principles, therefore they have a lot in common.

Principle 1: Customer Involvement

During the whole development process, customers should be intimately involved. Its purpose is to offer new system needs, priorities them and assess system iterations.

Principle 2: Incremental Delivery

The software is developed in steps with the client defining the needs for each iteration.

Principle 3: Focus on people do not process

This should recognize and utilize the development team's skills. Team members should be left without prescribed processes to create their own methods of functioning.

Principle 4: Embrace Change

To adapt to changes, consider the system requirements.

Principle 5: Focus on simplicity

Concentrate on simplicity in the development process and software. Work to eliminate the complexity of the system whenever feasible.

5.2.2 Agile Method Limitations

Limitations of agile methods :

- A) In reality, it is sometimes challenging to actualize the principles behind agile methods:
1. Although the notion of customer participation in development is intriguing, the success of this idea depends on a customer who can spend time in the development team and represent all the systems. Customer representatives are often under different demands and cannot participate fully in the creation of software.
 2. Individual team members may not have appropriate personalities that are typical of agile techniques for intensive engagement and consequently do not connect effectively with other team members.
 3. Changes may be exceedingly difficult to priorities, particularly in systems with numerous stakeholders. Typically, the varied priorities of each stakeholder are distinct.
 4. To maintain simplicity, more work is needed. The team members may not have time to make beneficial simplifications of systems under pressure from delivery schedules.
 5. Many organizations, particularly major corporations, have spent years transforming their culture, defining and following processes. They find

it challenging to transition to a working paradigm in which the procedures of development teams are informal and defined.

- B) Another non-technical difficulty – which is an overall problem of agile development and delivery – is when the system client hires a foreign system development company. Usually, the software requirement document is part of the customer-supplier contract.

Since the incremental specification is inherent in agile techniques, it may be challenging to write contracts for this kind of development.

Agile approaches must thus depend on developing a precise set of criteria on contracts in which the Customer pays for the time necessary to create the system. This helps the client and developer as long as everything goes smoothly. However, it may be difficult if issues occur who is guilty and who should pay for additional time and resources.

- C) The principal problem is that consumers are less likely to be involved after delivery of software. Although a client can justify a representative's complete engagement in the system development process, this is less probable when the modifications do not occur during maintenance. The system will certainly lose interest for the customer reps.
- D) The other difficulty is that the development crew will be continuous. Agile techniques are based on team members who grasp the system elements without consulting documentation. If an agile development team breaks down then implicit knowledge is lost and the development of the system and its components is challenging for new team members.

In the context of agile techniques and maintenance, two questions have to be considered:

1. Are systems built by means of an agile methodology maintainable, considering the importance of avoiding formal documentation in the development process?
2. Can agile approaches be utilized to develop a system successfully in response to client demands for change?

The system should be described by official documents, making it easier to comprehend for those who change the system. However, formal documentation is sometimes not kept up to date in reality and hence does not reflect the software correctly. Agile approaches therefore highlight the

Importance of well-structured code authoring and code enhancement investment efforts.

5.3 Plan-Driven and Agile Development

The design and execution of agile approaches to software development are the key tasks in the software process. They include additional activities in the design and execution, such as requirements elicitation and testing. In contrast, a plan-driven software engineering methodology recognizes different phases in the development process with each stage-specific outcome. As a foundation for the planification of succeeding process activities, the results from one phase are employed.

With a plan-driven approach, iteration takes place in activities with formal documentation utilized to communicate across process phases. The requirements are, for instance, changing and finally specifying the needs are generated. This then contributes to the design and execution process. In Agile methodology, iteration takes place through activities. The requirements and design are therefore created jointly and not individually.

A plan-driven software process can enable progressive development and delivery. The needs can precisely be allocated, and the design and development process may be planned as a whole. An agile approach does not necessarily focus on coding, and some design documentation can be produced. Indeed, most software projects incorporate practical techniques that are plan-based and agile.

5.3.1 Issues to Consider When Balancing Agile and Plan Based Development

You have to address a range of technical, human and organizational issues to select the balance between a plan-based strategy and an agile one:

1. Is the specification and design extremely comprehensive before we get to implementation important? If such is the case, you should probably utilize a plan-led approach.
2. Does the software provided to clients and quick feedback from them really work in an incremental delivery strategy? If so, take agile approaches into account.
3. How big is the development system? Agile approaches are most efficient when a small, co-located team can create the system and communicate informally. This may not be viable for big systems requiring larger development teams to adopt a plan-driven approach.
4. What kind of system is developed? To do this study systems that require much analysis before they are implemented (e.g., in real time systems with complicated schedule requirements). In such cases, a plan driven strategy may be ideal.
5. How long is the system life expectancy? Long-term systems may need additional design documentation to explain the system's original objectives.

6. Which technologies are accessible for the creation of the system? Agile approaches frequently depend on effective tools to track a changing design. You may need additional design documentation if you are creating a system using an IDE that doesn't offer excellent capabilities to see or analyze the software.
7. What is the organization of the development team? If the team is spread or part of development is outsourced, design papers may need to be developed to communicate with all development teams. You might have to plan what these are in advance.
8. Do cultural concerns impact the growth of the system? 8. Traditional engineering organizations, because this is the norm in engineering, have a culture of planning development. This generally calls for comprehensive design documentation instead of informal knowledge in agile procedures.
9. How excellent are the development team designers and programmers? It is occasionally stated that agile techniques need greater levels of competence than plans in which programmers simply translate a detail in coding. You may have to utilize the best individuals to produce the design with others in charge of programming if you have a team with relatively less skill levels.
10. Is the system regulated from outside? 10. If an external regulator must approve a system, you will probably have to provide extensive evidence in the event of system security.

In fact, it is not particularly necessary to consider whether a project can be characterized as plan-led or agile. Finally, consumers of a software system are primarily concerned about if they have an executable system that matches their demands and does things helpful to the individual user or the company. Many businesses which claim agile methodologies have practiced and really embraced some agile practices.

5.4 Extreme Programming

The requirements are defined as scenarios (called user stories) in extreme programming, which are immediately carried out as a series of activities. Programmers work in pairs before creating the code and write tests for each assignment. When the new code is incorporated into the system, all tests must be carried out successfully. The system releases have a limited time span.

Extreme programming involves a number of practices:

1. Small, regular system releases promote incremental development. Simple customer tales or scenarios are the basis for determining which features should be added in an increase in the system.

2. The ongoing engagement of the customer in the development team supports consumer participation. The client representative is responsible for defining acceptance tests for the system and participating in the development process.
3. The co-programming, community ownership and sustainable growth processes that do not require excessive working hours of persons will not support processes.
4. Change includes frequent customer system releases, first development tests, restructuring to avoid degeneration of the code, and continuing new functional integration.
5. Constant refactoring that improves code quality and the application of basic designs that do not anticipate changes in the system needlessly assist simplicity maintenance. In an XP process, client needs are defined and prioritized closely. The client of the system is instead part of the team and discusses scenarios with others. Together, a 'story card' is developed which covers consumer demands. This scenario will then be implemented in a future software version by the development team.

Extreme programming adopts a 'extreme' methodology for progressive growth. New software versions may be created more than once a day, and consumers can get releases every two weeks. Release times are never slipped away; if there are issues with development, the client will be contacted, and functionality will be deleted. When a programmer creates an existing system to produce a new version, all current automated testing and new functionality tests must be performed.

The new software construction is only allowed if all testing is successful. This is the basis for the following system iteration. A basic concept of conventional software engineering is to plan changes. In other words, you should anticipate future software modifications and design them to be implemented quickly.

However, this idea has been rejected by extreme programming on the grounds that changing is often a waste of time. It's not valuable to take time to make changes more generically in a software. The anticipated modifications are frequently never done and entirely other requests for changes might be made. The XP methodology acknowledges this, and modifications are made and that the programmed is reorganized when such changes occur.

5.4.1 Testing Xp

No system specification may be utilized by an external test team for system tests with incremental development. As a result, in contrast to plan-driven testing, some methods to incremental development have been highly informal. XP stresses the need of program testing to prevent some of the difficulties of testing and system validation. XP provides a testing technique which lowers the likelihood of introduction into the current system version of undetected faults.

The main characteristics of the XP test are:

1. Test first development
2. incremental scenario tests
3. user participation in and validation of the tests
4. the usage of automated frameworks for testing.

Test-first development is one of XP's most significant innovations. You write the tests before you create the code instead of developing a code and then building tests for that code. You may then run the test as your code is created and identify difficulties while developing it. Implicit written tests define both an interface and a behavioral specification for the feature that is being built.

Problems of interface misunderstandings and requirements are reduced. This technique may be employed in any process in which the system needs and the code implementing that requirement are clearly related. This relationship can be always seen in XP, as the story cards reflecting the needs are divided into tasks and the tasks are the main implementation unit.

In test-first development, the task implementers must completely grasp the requirements in order to build system tests. This requires clarification of ambiguities and omissions in the specification prior to implementation. In addition, the problem of 'test-lag' is also avoided. This can occur if the system developer works quicker than the testing device. Implementation continues to advance the testing, and there is a propensity to bypass testing to preserve the development pace.

For the test-first development automated testing is necessary. A system that allows for easy writing and submission of a collection of tests for execution is an automated test framework. Since testing is automated, a number of tests may be performed quickly and simply.

Test-first development and automated testing generally leads to the preparation and execution of a significant number of tests.

This method does not lead to comprehensive program testing though. Three reasons are available:

1. Programmers prefer testing over programming and occasionally take shortcuts when they write tests.
2. Incremental writing of some tests might be quite tough.
3. The completeness of a test set is difficult to evaluate. While several system tests may be conducted, the test set may not cover things fully.

5.4.2 Pair Programming

Another unique technique established in XP is that software developers work in pairs. In fact, they sit on the same desktop to build the program. The same pairings do not usually schedule together, however. Rather, pairs are dynamically established to work together during the development process. All team members work together.

There are a few advantages of using pair programming:

1. It promotes the concept that the system is jointly owned and accountable.

This mirrors the notion of egoless programming from Weinberg (1971), where the software belongs to the complete team, and individuals are not liable for code issues. The team is instead collectively responsible for fixing these challenges.
2. It serves as a method of informal inspection since at least two individuals examine each line of code. Code inspections and reviews have been highly successful in finding a high level of software failure. They take time to organize, though, and usually delay the development process. Although pair programming is less official than code inspections, it is considerably cheaper.
3. It helps to facilitate software enhancement refactoring. The challenge is to rebuild this for long-term benefit in the regular development environment. Any person who does refactoring code might be considered less efficient than someone who develops code.

5.5 Agile Project Management

The software project managers are primarily responsible for the project management, so that the software is delivered in time and within the intended project budget. They oversee the work of software developers and monitor the progress of software development. Project management is a typical technique based on the plan.

Managers set up a project plan detailing when the project should be delivered and who will work on the project achievements. In order to have a plan-based strategy, a manager needs a steady picture of all developments and development processes.

Nevertheless, agile techniques do not function properly if plan-based approach is used because in agile techniques, changes in software and requirements are the rule. Agile development must be managed properly so that the time and resources available to the team may be used to the fullest. This needs an alternative project management strategy that is suited for the progressive growth of agile methodologies and the special strengths.

Scrum approach to project management is a general agile method where it focuses on managing development iteratively. It does not prescribe any technical approaches to agile software engineering like pair programming or test-first development.

It may therefore be utilized to give the project a management framework with more technical, agile techniques, like XP.

5.5.1 Stages in Scrum

Scrum consists of three stages.

- a) Firstly, the planning outline phase in which the main project goals are defined and the software architecture designed.
- b) Following this, there is a series of sprinting cycles. Its essential part, especially the sprint cycles, is the innovative characteristic of Scrum. A Scrum Sprint is a planning unit that evaluates the work, selects development features and implements the software. At the end of a sprint, the stakeholders are provided with the finished functionality.
- c) Finally, the closing stage of a project completes the project, completes necessary documentation, such as system assistance frames and user manuals and evaluates the lessons gained from the project.

The key features of this process are:

1. Fixed sprints, usually 2-4 weeks, are available. It is the development of a system release in XP.
2. A product backlog is the beginning point for the planning, which is the list of projects to be carried out. This is examined and priority and risk are allocated during the evaluation phase of the sprint. At the start of each sprint, the customer is intimately involved and might suggest new needs or tasks.
3. The phase of selection includes the entire project team working with the client to choose the features and functionality that will be built during the sprint.
4. The team arranges to create the software after they have been agreed upon. Short daily meetings are held with each team member to assess progress and reprioritize work if required. The team is separated from the client and company during this stage and all communications are routed via the so-called 'Scrum master.' The Scrum Master's duty is to prevent external distractions from developing teams. It depends on the challenge and the team how the task is carried out.
5. Work carried out is assessed and submitted to interested parties at the end of the sprint. Then starts the following sprint cycle. Scrum is based on the concept that the entire team should be empowered to make decisions in order to eliminate the concept of 'project manager.'

'Scrum master' is a facilitator who schedules day-to-day meetings, monitors backlogs, records decisions and measures backlog progress and communicates with clients and management outside of the team.

The whole staff attends everyday meetings, exchange information during the meeting, explain their progress since the last meeting and the difficulties that have emerged.

So everyone in the team understands what is going on and may replan short-term work to deal with these difficulties if they occur. Everybody is involved in this short-term strategy – from the Scrum master there are no orders or instructions given.

Benefits of Scrum Methodology

1. The product is divided into a series of digestible and comprehensible pieces.
2. No progress can be held up by unstable needs.
3. Everything is visible to the entire team and so communication amongst the team is better.
4. Customers notice increments on time and receive feedback on the functioning of the product.
5. Confidence is formed between clients and developers and a good culture in which everybody expects the project to succeed is developed.

5.6 Scaling Agile Methods

Agile methods have been used by small programming teams who can work together in the same room and communicate informally. Accordingly agile approaches have mainly been utilized for small and medium-sized systems development.

Naturally, for larger systems, too, there is a need to produce software more quickly, which is more adapted for client demands. The scaling up of agile methodologies to deal with larger systems, built by huge businesses, has therefore been of significant interest.

The development of large software systems is distinct from the development of small systems in several ways:

1. Large systems generally consist of collections of discrete communication systems, which are developed by individual teams. These teams often operate at many locations, sometimes in various time zones. Every team cannot have a glimpse of the entire system effectively. Therefore, their priorities are normally to complete their system component without taking into account broader systemic concerns.

2. Political problems might also be important here, frequently changing an existing system is the quickest answer to a problem. This needs, however, negotiations with the system management to convince them that the modifications can be performed without risking the operation of the system.
3. A substantial percentage of the development involves systems configuration rather than the original code development, where multiple systems are incorporated into the design of a system. This does not necessarily match the progressive design and frequent integration of the system.
4. External laws and regulations which restrict the method in which large systems and their development procedures can be built, require specific kinds of system documentation to be produced, etc.
5. Long procurement and development times for large systems. It's hard to have coherent teams that know the system over this time, since individuals inevitably migrate to other jobs and projects.
6. Generally, large systems have several stakeholders. For instance, nursing staff and administrators may be the end users of a medical system, but the stakeholders include senior medical professionals, hospital management etc. All these many stakeholders cannot be included in the development process effectively.

The scalability of agile methodologies offers two perspectives:

- A. A 'scaled-up' approach that uses such methodologies to create big software systems that a small team cannot develop.
- B. A "scaling out" perspective, which concerns how agile approaches with many years of software development expertise may be adopted throughout a big business.
 1. For the development of big systems, you cannot concentrate on the system code alone. You need to develop and document your system more at an early stage. There must be software architecture and documentation generated in order to define important components of the system, such as database schemes, team breakdowns, etc.
 2. Mechanisms for cross-team communication must be developed and utilized. This should include regular talks with team members on telephones and videos and frequent, brief electronic meetings, when teams update each other on progress. To enable communication a range of communication channels should be offered, for example, e-mail, instant messaging, wikis and social networking platforms.
 3. Continuous integration is practiced when the entire system is built every time a developer reviews a modification. However, frequent

system builds, and regular system releases are important to maintain. This could lead to the introduction of new technologies for configuration management supporting multi-team software development.

5.6.1 Challenges In Implementation

Agile methodologies at large firms are challenging to implement for a number of reasons:

1. Project managers with no agile methodologies' expertise may be reluctant to take the risk of a new approach since they don't know how their projects will influence it.
2. Because of their bureaucracy, large companies frequently enforce quality processes and standards that are required to be followed by all projects. Sometimes software tools support these and the use of the tools is made compulsory.
3. Agile approaches appear to be most effective when team members have a reasonably high degree of competence. There are, however, a wide variety of talents and capabilities inside huge businesses, and those with lesser levels of expertise are probably not efficient team members in agile processes.
4. In particular in businesses with a history of utilizing conventional systems engineering, cultural opposition might exist to agile approaches. Examples of companies' procedures which may not be compatible with agile methodologies are change management and testing procedures.

Change management is the process of controlling changes to a system, so that the impact of changes is predictable, and costs are controlled. All modifications must be approved before being made and the concept of refactoring is in contradiction with it.

A process of cultural transformation is the introduction and maintenance of the use of agile methodologies across a major organisation. It takes a long time to implement cultural change and frequently requires a change in administration prior to its implementation.

5.7 Summary:

1. Agile techniques often rely on a progressive approach to the design, development, and delivery of software. They are ideal for the creation of an application software when the system must change fast in the process of development. That will give software to clients rapidly, who may later submit new and modified needs to be integrated in the system subsequently.

2. A plan-driven software process can enable progressive development and delivery. The needs can accurately be allocated, and the design and development process may be planned as a whole. An agile approach does not always focus on coding, and some design documentation can be produced. Indeed, most software projects incorporate practical techniques that are plan-based and agile.

You have to address a range of technical, human and organizational issues to select the balance between a plan-based strategy and an agile one.

3. Extreme programming adopts an 'extreme' methodology for progressive growth. New software versions may be created more than once a day, and consumers can get releases every two weeks. Release times are never slipped away; if there are issues with development, the client will be contacted, and functionality will be deleted.
4. Scrum approach to project management is a general agile method where it focuses on managing development iteratively. It does not prescribe any technical approaches to agile software engineering like pair programming or test-first development.

It may therefore be utilized in order to give the project a management framework with more technical, agile techniques, like XP. Scrum consists of three stages.

- a) Firstly, the planning outline phase in which the main project goals are defined and the software architecture designed.
- b) Following this, there is a series of sprinting cycles. Its essential part, especially the sprint cycles, is the innovative characteristic of Scrum. A Scrum Sprint is a planning unit that evaluates the work, selects development features and implements the software. At the end of a sprint, the stakeholders are provided with the finished functionality.
- c) Finally, the closing stage of a project completes the project, completes necessary documentation, such as system assistance frames and user manuals and evaluates the lessons gained from the project.

5.8 Questions

- 1) What is agile software development methodology?
- 2) When to use a plan driven approach and agile approach?
- 3) What is Extreme Programming?
- 4) What is pair programming?
- 5) Why is it necessary for larger businesses to use agile methodology?
- 6) What is the Scrum approach?

- 7) How is the development of smaller software systems different from large software systems?
- 8) What are the different technical, human and organisational issues that one has to consider while striking a balance between plan driven and agile approach?
- 9) What are the limitations of agile methods?
- 10) What are the benefits of scrum methodology?

5.9 References

<https://www.manage.gov.in/studymaterial/PM.pdf>

Software Engineering Edition, Ian Somerville, Pearson Education

Software Engineering, Pankaj Jalote, Narosa Publication

Software Engineering, A practitioner's approach, Roger Pressman, Tata McGraw Hill

<https://www.comakeit.com/blog/agile-software-development-approach-save-business-pandemic-disruptions/>

<https://www.agilealliance.org/agile101/>

SOCIO-TECHNICAL SYSTEM

Unit Structure

6.0 Objectives

6.1 Essential Characteristics of Socio technical systems

6.2 Emergent System Properties

6.3 Systems Engineering

6.4 Components of Systems such as organization, people and computers.

6.5 Dealing Legacy Systems

6.0 Objectives

At the end of this unit, the student will be able to

- To explain what a socio-technical system is and the distinction between this and a computer-based system
- To introduce the concept of emergent system properties such as reliability and security
- To explain system engineering and system procurement processes
- To explain why the organizational context of a system affects its design and use
- To discuss legacy systems and why these are critical to many businesses.

6.1 Essential Characteristics of Socio Technical Systems

1. The term system is one that is universally used. We talk about computer systems, operating systems, payment systems, the educational systems, the systems of government and so on.
2. These are all obviously quite different uses of the word system although they share the characteristic that, somehow, the system is more than simply the sum of its parts.
3. A useful working definition of these types of systems is: A system is a purposeful collection of interrelated components that work together to achieve some objective.
4. This general definition embraces a vast range of systems. For example, a very simple system such as a pen may only include three or four hardware components. By contrast, an air traffic control system includes thousands of

hardware and software components plus human users who make decisions based on information from the computer system.

5. Systems that include software fall in to two categories

5.1 Technical computer-based systems are systems that include hardware and software components but not procedures and processes. Examples of technical systems include televisions, mobile phones and most personal computer software.

5.2 Socio-technical systems include one or more technical systems but, crucially, also include knowledge of how the system should be used to achieve some broader objective. This means that these systems have defined operational processes, include people (the operators) as inherent parts of the system, are governed by organisational policies and rules and may be affected by external constraints such as national laws and regulatory policies.

6 Essential Characteristics of Socio-Technical Systems are as follows

6.1 They have emergent properties that are properties of the system as a whole rather than associated with individual parts of the system. Emergent properties depend on both the system components and the relationships between them. As this is so complex, the emergent properties can only be evaluated once the system has been assembled.

6.2 They are often nondeterministic. This means that, when presented with a specific input, they may not always produce the same output. The system's behaviour depends on the human operators, and people do not always react in the same way. Furthermore, use of the system may create new relationships between the system components and hence change its emergent behaviour.

6.3 The extent to which the system supports organisational objectives does not just depend on the system itself. It also depends on the stability of these objectives, the relationships and conflicts between organisational objectives and how people in the organisation interpret these objectives. New management may reinterpret the organisational objective that a system is designed to support, and a successful system may then become a failure.

7 Software engineers should have some knowledge of socio-technical systems and systems engineering (White, et al., 1993; Thayer, 2(02) because of the importance of software in these systems. For example, there were fewer than 10 megabytes of software in the US Apollo space program that put a man on the moon in 1969, but there are about 100 megabytes of software in the control systems of the Columbus space station.

- 8 A characteristic of all systems is that the properties and the behaviour of the system components are inextricably intermingled. The successful functioning of each system component depends on the functioning of some other components. Thus, software can only operate if the processor is operational.
- 9 Systems are usually hierarchical and so include other systems. For example, a police command and control system may include a geographical information system to provide details of the location of incidents. These other systems are called sub-systems.
- 10 A characteristic of sub-systems is that they can operate as independent systems in their own right. Therefore, the same geographical information system may be used in different systems.
- 11 Because software is inherently flexible, unexpected systems problems are often left to software engineers to solve. Say a radar installation has been sited so that ghosting of the radar image occurs. It is impractical to move the radar to a site with less interference, so the systems engineers have to find another way of removing this ghosting.
- 12 This situation, where software engineers are left with the problem of enhancing software capabilities without increasing hardware cost, is very common. A good example of this was the failure of the Denver airport baggage system (Swartz, 1996), where the controlling software was expected to deal with several limitations in the equipment used.
- 13 Software engineering is therefore critical for the successful development of complex, computer-based socio-technical systems. As a software engineer, you should not simply be concerned with the software itself but you should also have a broader awareness of how that software interacts with other hardware and software systems and how it is supposed to be used.
- 14 This knowledge helps us understand the limits of software, to design better software and to participate as equal members of a systems engineering group.

6.2 Emergent System Properties

1. The complex relationships between the components in a system mean that the system is more than simply the sum of its parts. It has properties that are properties of the system as a whole.
2. These emergent properties (Checkland, 1981) cannot be attributed to any specific part of the system. Rather, they emerge only once the system components have been integrated. Some of these properties can be derived directly from the: comparable properties of sub-systems.

- 3 However, more often, they result from complex sub-system interrelationships that cannot, in practice, be derived from the properties of the individual system components. Examples of some emergent properties are shown in table 1 given below.
- 4 There are two types of emergent properties
 - 4.1 Functional emergent properties appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.
 - 4.2 a) Non-functional emergent properties relate to the behaviour of the system in its operational environment. Examples of non-functional properties are reliability, performance, safety and security. These are often critical for computer-based systems, as failure to achieve some minimal defined level in these properties may make the system unusable. b) Some users may not need some system function, so the system may be acceptable without them. However, a system that is unreliable or too slow is likely to be rejected by all its users.
- 5 To illustrate the complexity of emergent properties, consider the property of system reliability. Reliability is a complex concept that must always be considered at the system level rather than at the individual component level. The component in a system are interdependent, so failures in one component can be propagated through the system and affect the operation of other component failures propagate through the system.
6. It is often difficult to anticipate how the consequences of component failures propagate through the system. Consequently, you cannot make good estimates of overall system reliability from data about the reliability of system components.
- 7 There are three related influences on the overall reliability of a system
 - 7.1 Hardware reliability What is the probability of a hardware component failing and how long does it take to repair that component?
 - 7.2 Software reliability How likely is it that a software component will produce an incorrect output? Software failure is usually distinct from hardware failure in that software does not wear out. Failures are usually transient so the system carries on working after an incorrect result has been produced
 - 7.3 Operator reliability How likely is it that the operator of a system will make an error?

- 8 The software can then behave unpredictably. Operator error is most likely in conditions of stress, such as when system failures are occurring. These operator errors may further stress the hardware, causing more failures, and so on.
- 9 Thus, the initial, recoverable failure can rapidly develop into a serious problem requiring a complete system shutdown.
- 10 Like reliability, other emergent properties such as performance or usability are hard to assess but can be measured after the system is operational. Properties such as safety and security, however, pose different problems.
- 11 A secure system is one that does not allow unauthorised access to its data but it is clearly impossible to predict all possible modes of access and explicitly forbid them. Therefore, it may only be possible to assess these properties by default. That is, you only know that a system is insecure when someone breaks into it.
- 12 Table 1 Examples of Emergent Properties

Sr.No	Property	Description
1	Volume	The volume of a system varies depending on how the component assemblies are arranged and connected
2	Reliability	System reliability depends on component reliability but unexpected interactions can cause new types of failure and therefore affect the reliability of the system
3	Security	The security of the system is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguard.
4	Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty, modify or replace these components.
5	Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators and its operating environment.

6.3 Systems Engineering

1. Systems engineering is the activity of specifying, designing, implementing, validating, deploying and maintaining socio-technical systems.
2. Systems engineers are not just concerned with software but also with hardware and the system's interactions with users and its environment. They must think about the services that the system provides, the constraints under which the system must be built and operated and the ways in which the system is used to fulfil its purpose.
3. software engineers need an understanding of system engineering because problems of software engineering are often a result of system engineering decisions.

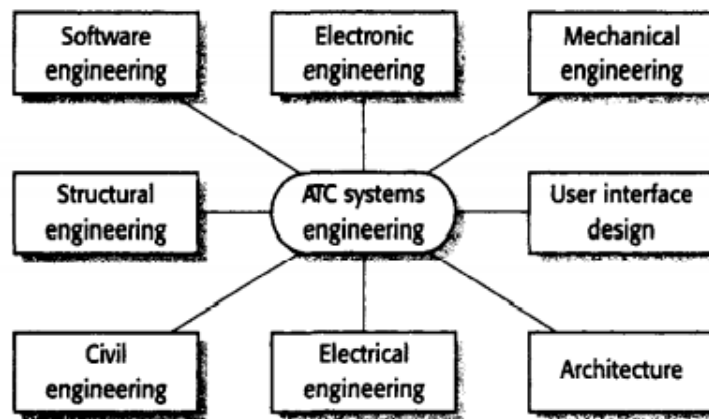


Fig 1 Disciplines involved in Systems Engineering

4. There are important distinctions between the system engineering process and the software development process
 - 4.1 Limited scope for rework during system development Once some system engineering decisions, such as the siting of base stations in a mobile phone system, have been made, they are very expensive to change. Reworking the system design to solve these problems is rarely possible. One reason software has become so important in systems is that it allows changes to be made during system development, in response to new requirements.
 - 4.2 Interdisciplinary involvement Many engineering disciplines may be involved in system engineering. There is a lot of scope for misunderstanding because different engineers use different terminology and conventions.

- 5 Systems engineering is an interdisciplinary activity involving teams drawn from various backgrounds. System engineering teams are needed because of the wide knowledge required to consider all the implications of system design decisions. Fig 1 shows some of the disciplines that may be involved in the system engineering team for an air traffic control (ATC) system that uses radars and other sensors to determine aircraft position.
- 6 For many systems, there are almost infinite possibilities for trade-offs between different types of sub-systems. Different disciplines negotiate to decide how functionality should be provided. Often there is no correct' decision on how a system should be decomposed.
- 7 Rather, you may have several possible alternatives, but you may not be able to choose the best technical solution. Say one alternative in an air traffic control system is to build new radars rather than refit existing installations.
- 8 If the civil engineers involved in this process do not have much other work, they may favour this alternative because it allows them to keep their jobs. They may then rationalise this choice with technical arguments.

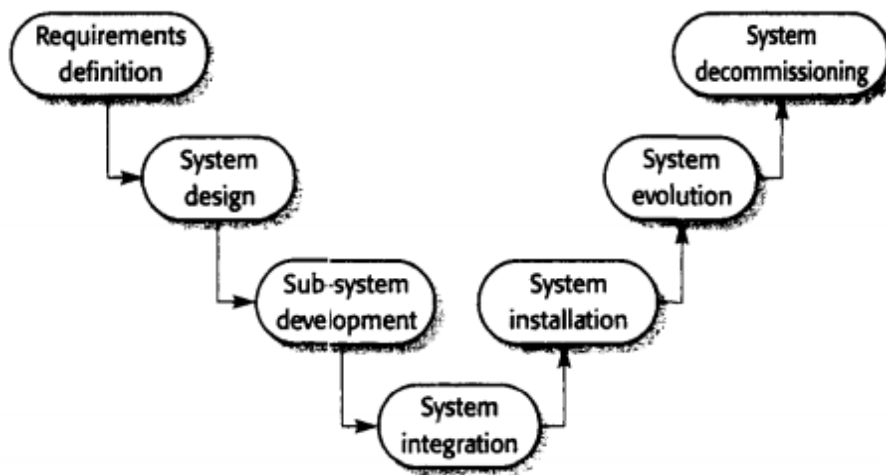


Fig 2 System Engineering Process

- 9 The above figure explains the phases of the systems engineering process. This process was an important influence on the 'waterfall' model of the software process.
- 10 Types of System Engineering
 - 10.1 System requirement definition
 1. System requirements definitions specify what the system should do (its functions) and its essential and desirable system properties. As with software requirements analysis.

2. As with software requirements analysis creating system requirements definitions involves consultations with system customers and end-users. This requirements definition phase usually concentrates on deriving three types of requirements
 - 2.1 Abstract functional requirements the basic functions that the system must provide are defined at an abstract level. More detailed functional requirements specification takes place at the sub-system level. For example, in an air traffic control system, an abstract functional requirement would specify that a flight-plan database should be used to store the flight plans of all aircraft entering the controlled airspace. However, you would not normally specify the details of the database unless they affected the requirements of other sub-systems.
 - 2.2 System properties These are non-functional emergent system properties such as availability, performance and safety. These non-functional system properties affect the requirements for all sub-systems.
 - 2.3 Characteristics that the system must not exhibit It is sometimes as important to specify what the system must not do as it is to specify what the system should do. For example, if you are specifying an air traffic control system, you might specify that the system should not present the controller with too much information.
- 3 An Important part of the requirements definition phase is to establish a set of overall objectives that the system should meet. These should not necessarily be expressed in terms of the system's functionality but should define why the system is being procured for a particular environment.
- 4 For eg specifying a system for an office building to provide for fire protection and for intruder detection. A statement of objectives based on the system functionality might be to provide a fire and intruder alarm system for the building that will provide internal and external warning office or unauthorised intrusion.
- 5 This objective states explicitly that there needs to be an alarm system that provides warnings of undesired events. Such a statement might be appropriate if you were replacing an existing alarm system. By contrast, a broader statement of objectives might be to ensure that the normal functioning of the work carried out in the building is not seriously disrupted by events such as fire and unauthorized intrusion.

- 6 A fundamental difficulty in establishing system requirements is that the problems that complex systems are usually built to help tackle are usually 'wicked problems'. A wicked problem is a problem that is so complex and where there are so many related entities that there is no definitive problem specification.

10.2 System Design

- 1 System design is concerned with how the system functionality is to be provided by the components of the system. The activities involved in this process are
 - 1.1 Partition requirements You analyse the requirements and organise them into related groups. There are usually several possible partitioning options, and you may suggest a number of alternatives at this stage of the process.
 - 1.2 Identify sub-systems You should identify sub-systems that can individually or collectively meet the requirements. Groups of requirements are usually related to sub-systems, so this activity and requirements partitioning may be amalgamated. However, sub-system identification may also be influenced by other organisational or environmental factors.
 - 1.3 Assign requirements to sub-systems You assign the requirements to subsystems. In principle, this should be straightforward if the requirements partitioning is used to drive the sub-system identification. In practice, there is never a clean match between requirements partitions and identified sub-systems. Limitations of externally purchased sub-systems may mean that you have to change the requirements to accommodate these constraints.
 - 1.4 Specify sub-system functionality You should specify the specific functions provided by each sub-system. This may be seen as part of the system design phase or, if the sub-system is a software system, part of the requirements specification activity for that system. You should also try to identify relationships between sub-systems at this stage.
 - 1.5 Define sub-system interfaces You define the interfaces that are provided and required by each sub-system. Once these interfaces have been agreed upon, it becomes possible to develop these sub-systems in parallel.

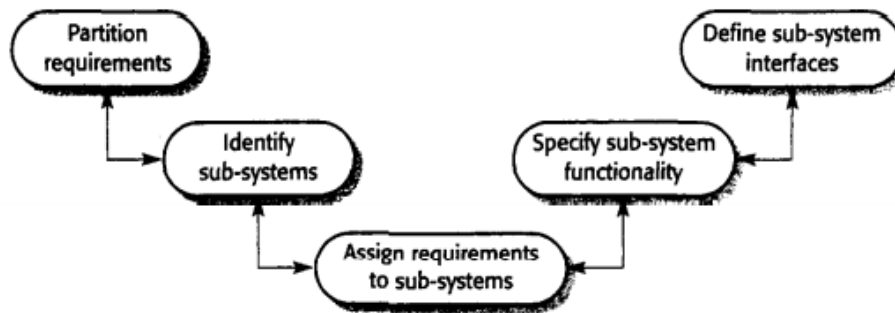


Fig 2 The System design process

- 2 The spiral process reflects the reality that requirements affect design decisions and vice versa and so it makes sense to interleave these processes. Starting in the centre, each round of the spiral may add detail to the requirements and the design. Some rounds may focus on requirements, some on design. Sometimes, new knowledge collected during the requirements and design process means that the problem statement itself has to be changed.

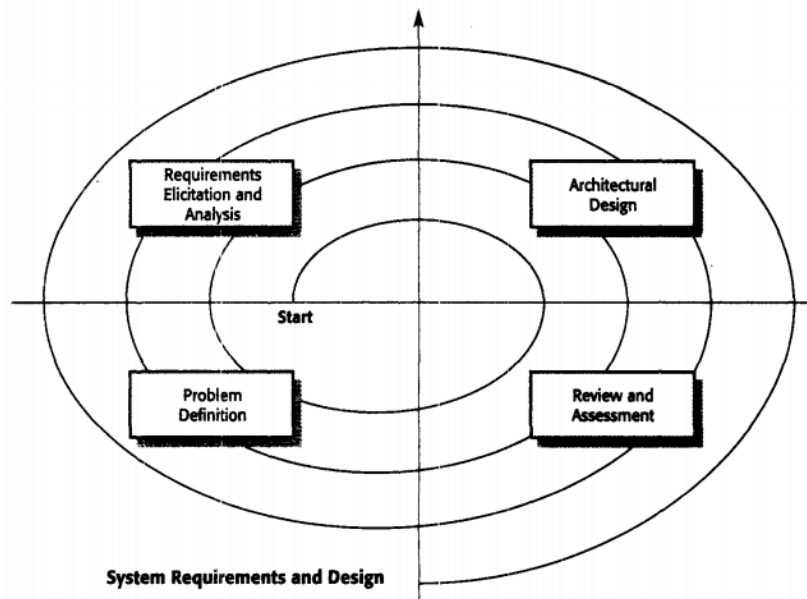


Fig 4 A spiral model of requirements and design

10.3 System Modelling

1. During the system requirements and design activity, systems may be modelled as a set of components and relationships between these components. The system architecture may be presented as a block diagram showing the major sub-systems and the interconnections between these sub-systems.

- 2 When drawing a block diagram, you should represent each sub-system using a rectangle, and we should show relationships between the sub-systems using arrows that link these rectangles. The relationships indicated may include data flow, is used by' relationship or some other type of dependency relationship.
- 3 For eg figure shown below depict the decomposition of an intruder alarm system in to its principal components.

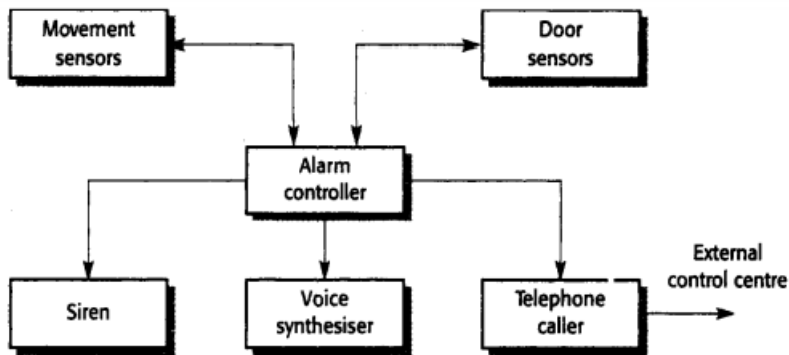


Fig 5 A simple Burglar Alarm System

- 4 Every component of above figure is explained in table 2 given below

Sr. No	Subsystem	Description
1	Movement Sensors	Detects movement in the rooms monitored by the system
2	Door Sensors	Detects door opening in the external doors of the building
3	Alarm Controller	Controls the operation of the system
4	Siren	Emits an audible warning when an intruder is suspected
5	Voice Synthesizer	Synthesises a voice message giving the location of the suspected intruder
6	Telephone Caller	Makes external calls to notify security, the police etc

- 5 Figure shown below the architecture of a much larger system for air traffic control. Several major sub-systems shown are themselves large systems. The arrowed lines that link these systems show information flow between these sub-systems.

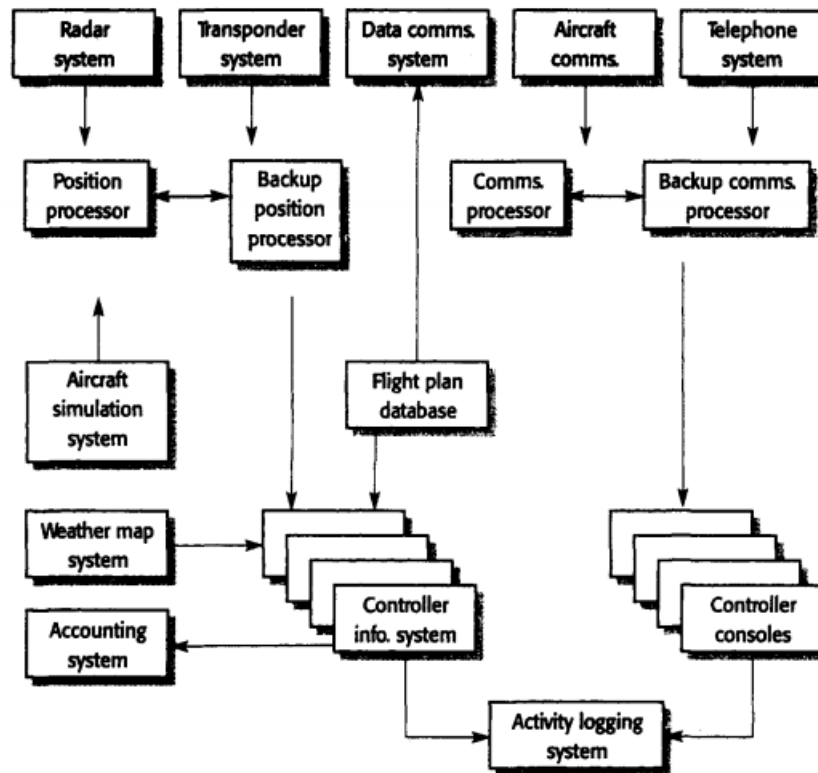


Fig 6 An architectural model of an air traffic control system

10.4 Sub-System development

- 1 During sub-system development, the sub-systems identified during system design are implemented. This may involve starting another system engineering process for sub-system is software, a software process involving requirements, design, implement.
- 2 Occasionally, all sub-systems are developed from scratch during the development process. Normally, however, some of the sub-systems are commercial, off-the-shelf (COTS) systems that are bought for integration into the system.
- 3 Sub-systems are usually developed in parallel. When problems are encountered that cut across sub-system boundaries, a system modification request must be made. Where systems involve extensive hardware engineering, making modifications after manufacturing has started is usually very expensive.

10.5 System Integration

- 1 During the systems integration process, you take the independently developed sub systems and put them together to make up a complete

system. Integration can be done using a 'big bang' approach, where all the sub-systems are integrated at the same time.

- 2 However, for technical and managerial purposes, an incremental integration process where sub-systems are integrated one at a time is the best approach, for two reasons as follows
 - 2.1 It is usually impossible to schedule the development of all sub-systems so that they are all finished at the same time.
 - 2.2 a) Incremental integration reduces the cost of error location. If many sub-systems are simultaneously integrated, an error that arises during testing may be in any of these sub systems. b) When a single sub-system is integrated with an already working system, errors that occur are probably in the newly integrated sub-system or in the interactions between the existing subsystems and the new sub-system.
- 3 Once the components have been integrated, an extensive programme of system testing takes place.
- 4 Sub-system faults that are a consequence of invalid assumptions about other subsystems are often revealed during system integration. This may lead to disputes between the various contractors responsible for the different sub-systems.
- 5 As more and more systems are built by integrating COTS hardware and software components, system integration is becoming increasingly important.

10.6 System Evolution

- 1 Large, complex systems have a very long lifetime. During their life, they are changed to correct errors in the original system requirements and to implement new requirements that have emerged. The organisation that uses the system may reorganise itself and hence use the system in a different way. The external environment of the system may change, forcing changes to the system.
- 2 System evolution is like software evolution is inherently costly for several reasons
 - 1 Proposed changes have to be analysed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.
 - 2 Because sub-systems are never completely independent, changes to one subsystem may adversely affect the performance or

behaviour of other subsystems. Consequent changes to these subsystems may therefore be needed.

- 3 The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
- 4 As systems age, their structure typically becomes corrupted by change so the costs of making further changes increases.

6.4 Components of Systems Such As Organization, People And Computers.

1. Socio-technical systems are enterprise systems that are intended to help deliver some organisational or business goal. This might be to increase sales, reduce material used in manufacturing, collect taxes, maintain a safe airspace, etc.
- 2 Because they are embedded in an organisational environment, the procurement, development and use of these system is influenced by the organisation's policies and procedures and by its working culture.
- 3 The users of the system are people who are influenced by the way the organisation is managed and by their interactions with other people inside and outside of the organisation.
- 4 Therefore, when we are trying to understand the requirements for a socio-technical system we need to understand its organisational environment. If we don't understand the systems may not meet business needs, and users and their managers may reject the system.
- 5 Human and organisational factors from the system's environment that affect the system design include as follows
 - 5.1 Process changes Does the system require changes to the work processes in the environment? If so, training will certainly be required. If changes are significant, or if they involve people losing their jobs, there is a danger that the users will resist the introduction of the system.
 - 5.2 Job changes Does the system de-kill the users in an environment or cause them to change the way they work? If so, they may actively resist the introduction of the system into the organisation. Designs that involve managers having to change their way of working to fit the computer system are often resented. The managers may feel that their status in the organisation is being reduced by the system.
 - 5.3 Organisational changes Does the system change the political power structure in an organisation? For example, if an organisation is

dependent on a complex system, those who know how to operate the system have a great deal of political power.

6 Organizational Processes

1. The development process is not the only process involved in systems engineering. It interacts with the system procurement process and with the process of using and operating the system. This depicts in figure given below

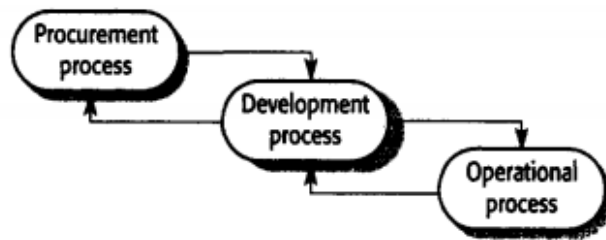


Fig7 Procurement, development and operational processes

- 2 The procurement process is normally embedded within the organisation that will buy and use the system (the client organisation). The process of system procurement is concerned with making decisions about the best way for an organisation to acquire a system and deciding on the best suppliers of that system.
- 3 Large complex systems usually consist of a mixture of off-the-shelf and specially built components. One reason why more and more software is included in systems is that it allows more use of existing hardware components, with the software acting as a 'glue' to make these hardware components work together effectively.
- 5 Figure given below depicts the procurement process for both existing systems and systems teams that have to be specially designed. Some important points about the process shown in the diagram are as follows
 - 5.1 Off the shelf components do not usually much requirements exactly, unless the requirements have been written with these components in mind. Therefore, choosing a system means that we have to find the closest match between the system requirements and the facilities offered by off the shelf systems.
 - 5.2 When a system is to be built specially, the specification of requirements acts as the basis of a contract for the system procurement. It is therefore a legal, as well as a technical document.

- 5.3 After a contractor to build a system has been selected, there is a contract negotiation period where we may have to negotiate further changes to the requirements and also discussed the issues such as the cost of changes to the system.

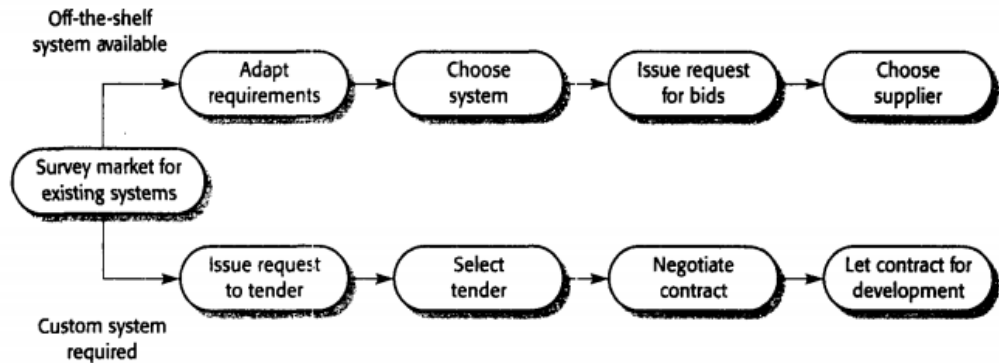


Fig 8 The System procurement process

- 6 The supplier who is usually called the principal contractor may contract out the development of different sub-systems to a number of sub-contractors. For large systems, such as air traffic control systems, a group of suppliers may form a consortium to bid for the contract.
- 7 The procurer deals with the contractor rather than the sub-contractors so that there is a single procurer/supplier interface. The sub-contractors design and build parts of the system to a specification that is produced by the principal contractor.
- 8 Operational processes are the processes that are involved in using the system for its defined purpose. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs and so on.
- 9 The key benefit of having people in a system is that people have a unique capability of being able to respond effectively to unexpected situations even when they have never had direct experience of these situations.
- 10 Designers should design operational processes to be flexible and adaptable. The operational processes should not be too constraining, they should not require operations to be done in a particular order, and the system software should not rely on a specific process being followed.
- 11 An issue that may only emerge after the system goes into operation is the problem of operating the new system alongside existing systems. There may be physical problems of incompatibility, or it may be difficult to transfer data from one system to another.

6.5 Dealing Legacy Systems

1. The time and effort required to develop a complex system, large computer-based usually have a long lifetime. For example, military systems are often designed for a 20 year lifetime and much of the world's air traffic control still relies on software and operational processes that were originally developed in the 1960's and 1970's.
2. Their development continues throughout their life with changes to accommodate new requirements, new operating platforms, and so forth.
3. Legacy systems are socio-technical computer-based systems that have been developed in the past, often using older or obsolete technology. These systems include not only hardware and software but also legacy processes and procedures-old ways of doing things that are difficult to change because they rely on legacy software.
4. Legacy systems are often business-critical systems. They are maintained because it is too risky to replace them. For example, for most banks the customer accounting system was one of their earliest systems. Organisational policies and procedures may rely on this system. If the bank were to scrap and replace the customer accounting software (which may run on expensive mainframe hardware) then there would be a serious business risk if the replacement system didn't work properly.
5. Figure shown below depicts the logical parts of a legacy system and their relationships
 - 5.1 System hardware-In many cases" legacy systems have been written for mainframe hardware that is no longer available, that is expensive to maintain and that may not be compatible with current organisational IT purchasing policies.
 - 5.2 Support software-The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.
 - 5.3 Application software- The application system that provides the business services is usually composed of a number of separate programs that have been developed at different times. Sometimes the term legacy system means this application software system rather than the entire system.
 - 5.4 Application data -These are the data that are processed by the application system. In many legacy systems, an immense volume of

data has accumulated over the lifetime of the system. This data may be inconsistent and may be duplicated in several files.

- 5.5 Business processes -These are processes that are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting all order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.
- 5.6 Business policies and rules -These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

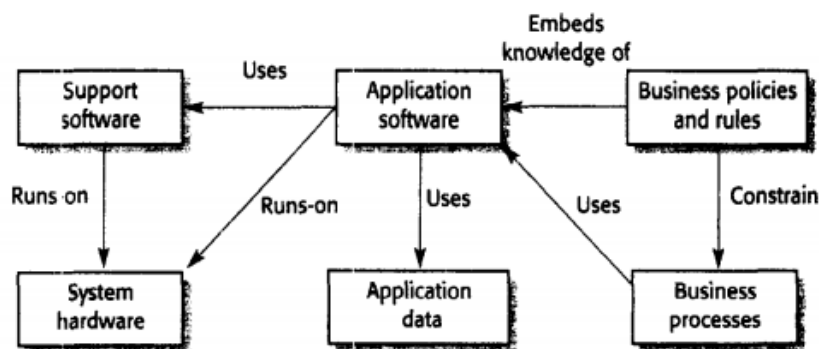


Fig 9 Legacy System Components

- 6 An alternative way of looking at these components of a legacy system is as a series of layers as shown in figure below. Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then we should be able to make changes within a layer without affecting either of the adjacent layers.

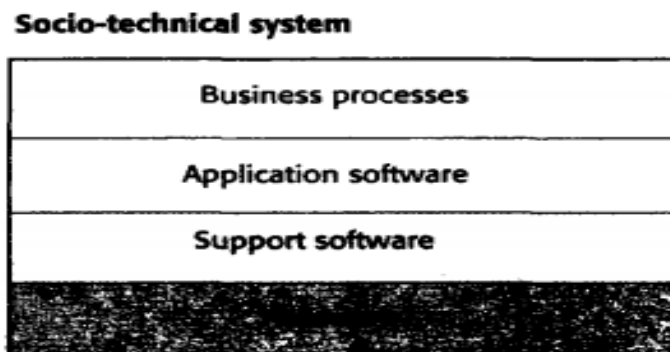


Fig 10 Layered model of a Legacy System

Questions

- Q1 Explain what is socio-technical system?
- Q2 Describe the emergent properties of socio-technical system>
- Q3 Illustrate the System engineering process along with-it types?
- Q4 Describe about the need of Organizational process in socio-technical system?
- Q5 Describe in short about Legacy Systems?

CRITICAL SYSTEMS

Unit Structure

- 7.0 Objectives
- 7.1 Types of Critical System
- 7.2 A Simple safety critical systems
- 7.3 Dependability of a system
- 7.4 Availability and Reliability
- 7.5 Safety and security of Software Systems

7.0 Objectives

At the end of this unit, the student will be able to

- Understand that in a critical system, system failure can have severe human or economic consequences.
- Illustrate four dimensions of system dependability- Availability, reliability, safety and security.
- Understand the importance of dependability with respect to software engineering.

7.1 Types of Critical System

- 1 Software failures are relatively common. In most cases, these failures cause inconvenience but no serious, long-term damage. However in some systems failure can result in significant economic losses, physical damage or threats to human life. Such type of systems are called critical systems.
- 2 Critical systems are technical or socio-technical systems that people or businesses depend on. If these systems fail to deliver their services as expected then serious problems and significant losses may result.
3. There are three main types of critical system
 - 3.1 Safety-critical systems -A system whose failure may result in injury, loss of life or serious environmental damage. An example of a safety-critical system is a control system for a chemical manufacturing plant.

- 3.2 Mission-critical systems -A system whose failure may result in the failure of some goal-directed activity. An example of a mission-critical system is a navigational system for a spacecraft.
- 3.3 Business-critical systems- A system whose failure may result in very high costs for the business using that system. An example of a business-critical system is the customer accounting system in a bank.
- 4 The most important emergent property of a critical system is its dependability. The term dependability was proposed by Laprie(1995) to cover the related systems attributes of availability, reliability, safety and security.
- 5. There are several reasons why dependability is the most important emergent property for critical systems.
 - 5.1 Systems that are unreliable, unsafe or insecure are often rejected by their users. If users don't trust a system, they will refuse to use it.
 - 5.2 System failure costs may be enormous. For some applications, such as a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system.
 - 5.3 Untrustworthy systems may cause information loss. Data is very expensive to collect and maintain, it may sometimes be worth more than the computer system on which it is processed.
- 6 Consequently, critical systems are usually developed using well-tried techniques rather than newer techniques that have not been subject to extensive practical experience.
- 7 Expensive software engineering techniques that are not cost-effective for noncritical systems may sometimes be used for critical systems development.
- 8 One reason why these formal methods are used is that it helps reduce the amount of testing required. For critical systems, the costs of verification and validation are usually very high-more than 50% of the total system development costs.
- 9 Although a small number of control systems may be completely automatic, most critical systems are socio-technical systems where people monitor and control the operation of computer-based systems.
- 10 There are three system components where critical systems failures may occur

- 10.1 System hardware may fail because of mistakes in its design, because components fail as a result of manufacturing errors or because the components have reached the end of their natural life.
- 10.2 System software may fail because of mistakes in its specification, design or implementation.
- 10.3 Human operators of the system may fail to operate the system correctly. As hardware and software have become more reliable, failures in operation are now probably the largest single cause of system failures.
- 11 These failures can be interrelated. A failed hardware component may mean system operator have to cope with an unexpected situation and additional workload.
- 12 As a result, it is particularly important that designers of critical systems take a holistic systems perspective rather than focus on a single aspect of the system. If the hardware, software and operational processes are designed separately without taking the potential weaknesses of other parts of the system take in to account, then it is more likely that errors at interfaces between the various parts of the system.

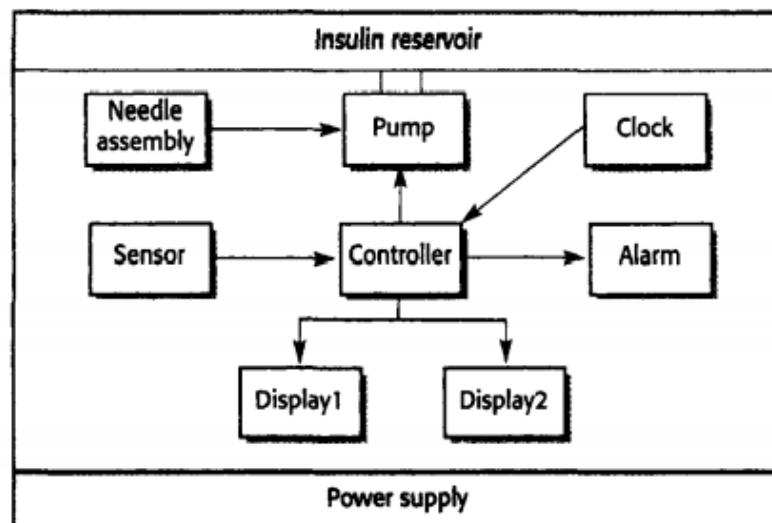


Fig 1 Insulin Pump Structure

7.2 A Simple Safety Critical Systems

1. There are many types of critical computer-based systems, ranging from control systems for devices and machinery to information and e-commerce systems. Understanding the critical system can be very difficult as we need

to understand the features and constraints of the application domain where they operate.

- 2 To understand the safety critical system, a simple example is taken over here is medical system that simulates the operation of the pancreas(internal organ). Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.
- 3 The problem with this treatment is that the level of insulin in the blood does not just depend on the blood glucose level but is a function of the time when the insulin injection was taken. This can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood sugar is, in the short term, a more serious condition, as it can result in temporary brain malfunctioning and, ultimately, unconsciousness and death. In the long term, continual high levels of blood sugar can lead to eye damage, kidney damage, and heart problems.
- 4 Current advances in developing miniaturised sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery system already exist for the treatment of hospitals patients.
- 5 A software-controlled insulin delivery system might work by using a micro-sensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a pump to deliver the insulin via a permanently attached needle.
- 6 Figure shown below depict the data flow model for transformation of insulin to a human body. There are two high-level dependability requirements for this insulin pump system
 - 6.1 The system shall be available to deliver insulin when required.
 - 6.2 The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

7. This might be possible that if system fails then human body gets excessive amount of insulin which threaten the life of the user.

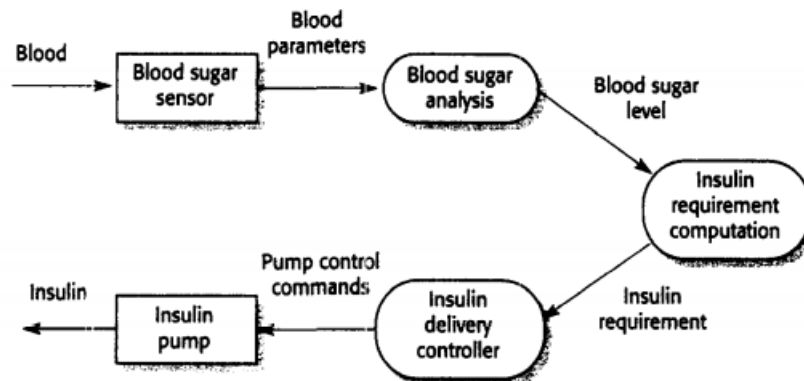


Fig 2 Data-flow model of the insulin pump

7.3 Dependability of A System

1. All of us are familiar with the problem of computer system failure. Without any reason computer systems sometimes crash and fail to deliver the services that have been requested.
2. Program running on these computers may not operate as expected and occasionally, may corrupt the data that is managed by the system.
3. We have learned to live with these failures and few of us completely trust the personnel computers that we normally use.
4. The dependability of a computer system is a property of the system that equates to its trustworthiness. Trustworthiness essentially means the degree of user confidence that the system will operate as they expect and that the system will not fail in normal use.
5. This property cannot be expressed numerically, but we use relative terms such as not dependable, very dependable and ultra dependable to reflect the degrees of trust that we have with system.
6. There are four principal dimensions to dependability as shown in figure below
 - 6.1 Availability- It is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.
 - 6.2 Reliability- It is the probability over a given period of time, that the system will cause damage to people or its environment.

- 6.3 Safety – It is a judgement of how likely it is that the system will cause damage to a people or its environment.
- 6.4 Security- It is a judgment of how likely it is that the system can resist accidental or deliberate intrusions.

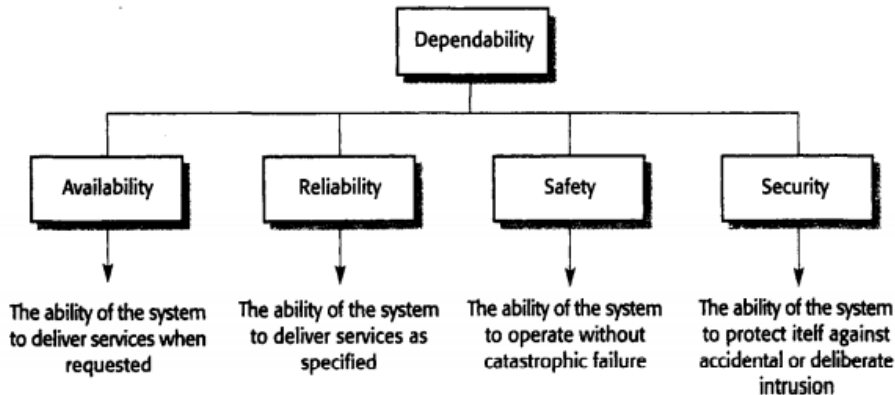


Fig 3 Dimensions of Dependability

- 7 These are complex properties that can be decomposed in to a number of other, simpler properties. For example security includes integrity and confidentiality. Reliability includes correctness (ensuring the system services are as specified), precision (ensuring information is delivered at an appropriate level of detail) and timeliness (ensuring that information is delivered when it is required).
- 8 The dependability properties of availability, security, reliability and safety are all interrelated. Safe system operation usually depends on the system being available and operating reliability. A system may become unreliable because its data has been corrupted by an intruder.
- 9 Denial-of-service attacks on a system are intended to compromise its availability. If a system that has been proved to be safe is infected with a virus, safe operation can no longer be assumed. It is because of these close links that the notion of system dependability as an encompassing property was introduced.
- 10 Properties of Dependability
- 10.1 Repairability- System failures are inevitable, but the disruption caused by failure can be minimised if the system can be repaired quickly. In order for that to happen, it must be possible to diagnose the problem, access the component that has failed and make changes to fix that component. Repairability in software is enhanced when the

organisation using the system has access to the source code and has the skills to make changes to it.

- 10.2 Maintainability- As systems are used, new requirements emerge. It is important to maintain the usefulness of a system by changing it to accommodate these new requirements. Maintainable software is software that can be adapted economically to cope with new requirements and where there is a low probability that making changes will introduce new errors into the system.
- 10.3 Survivability- A very important attribute for Internet-based systems is survivability, which is closely related to security and availability (Ellison, et al., 1999). Survivability is the ability of a system to continue to deliver service whilst it is under attack and, potentially, while part of the system is disabled. Work on survivability focuses on identifying key system components and ensuring that they can deliver a minimal service. Three strategies are used to enhance survivability-namely, resistance to attack, attack recognition and recovery from the damage caused by an attack.
- 10.4 Error Tolerance- This property can be considered as part of usability and reflects the extent to which the system has been designed so that user input error are avoided and tolerated. When user errors occur, the system should, as far as possible, detect these errors and either fix them automatically or request the user to re-input their data.
- 11 All project will not have all these dimensions dependability properties. For eg insulin pump system consist of properties like availability, reliability and safety whereas security is not required.
- 12 Dependable software includes extra, often redundant, code to perform the necessary checking for exceptional system states and to recover from system faults. This reduces system performance and increases the amount of store required by the software. It also adds significantly to the costs of system development.
- 13 Because of additional design, implementation and validation costs, increasing the dependability of a system can significantly increase development costs.
- 14 Figure given below shows relationship between costs and incremental improvements in dependability. The higher the dependability that you need, the more that you have to spend on testing to check that we have reached that

level. the exponential nature of this cost/dependability curve, it is not possible to demonstrate that a system is 100% dependable, as the costs of dependability assurance would then be infinite.

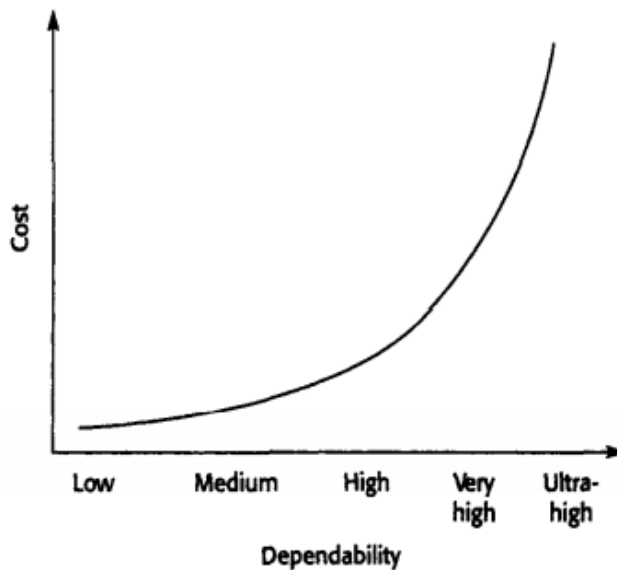


Fig 4 Cost/ Dependability Curve

7.4 Availability and Reliability

1. System availability and reliability are closely related properties that can both be expressed as numerical probability. The reliability of a system is the probability that the system's services will be correctly delivered as specified. The availability of a system is the probability that the: system will be up and running to deliver these services to users when they request them.
2. For example, some systems can have a high availability requirement but a much lower reliability requirement. If users expect continuous service, then the availability requirements are high. However, if the consequences of a failure are minimal and the system can recover quickly from these failures then the same system can have low reliability requirements.
3. An example of a system where availability is more critical than reliability is a telephone exchange switch. Users expect a dial tone when they pick up a phone so the system has high availability requirements. However, if a system fault causes a connection to fail, this is often recoverable.
4. Availability does not simply depend on the system itself but also on the time needed to repair the faults that make the system unavailable. Therefore, if

system A fails once per year, and system B fails once per month, then A is clearly more reliable than B.

- 5 System reliability and availability may be defined more precisely as follows
 - 5.1 Reliability-The probability of failure-free operation over a specified time in a given environment for a specific purpose.
 - 5.2 Availability- The probability that a system at a point in time, will be operational and able to deliver the requested services.
- 6 The definition of reliability states that the environment in which the system is used and the purpose that it is used for must be taken into account. For example, let's say that we measure the reliability of a word processor in an office environment where most users are uninterested in the operation of the software. They follow the instructions for its use and do not try to experiment with the system.
- 7 Human perceptions and patterns of use are also significant. For example, say a car has a fault in its windscreen wiper system that results in intermittent failures of the wipers to operate correctly in heavy rain. The reliability of that system as perceived by a driver depends on where they live and use the car.
- 8 A strict definition of reliability relates the system implementation to its specification. That is, the system is behaving reliably if its behaviour is consistent with that defined in the specification. Reliability and availability are compromised by system failures. These may be a failure to provide a service, a failure to deliver a service as specified, or the delivery of a service in such a way that is unsafe or insecure.
- 9 Human errors do not inevitably lead to system failures. The faults introduced may be in parts of the system that are never used. Faults do not necessarily result in system errors, as the faulty state may be transient and may be corrected before erroneous behaviour occurs.
- 10 Table shown below differentiate among the terms fault, error and failure

Sr.No	Term	Description
1	System Failure	An event that Occurs at some point in time when the system does not deliver a service as expected by its users
2	System Error	An erroneous system state that can lead to system behaviour.

3	System Fault	A characteristic of a software system that can lead to a system error. For example, failure to initialise a variable could lead to that variable having the wrong value when it is used.
4	Human Error or Mistake	Human behaviour that results in the introduction of faults in to a system

11 Three approaches that are used to improve the reliability of a system

11.1 Fault Avoidance-Development techniques are used that either minimise the possibility of mistakes and/or that trap mistakes before they result in the introduction of system faults. Examples of such techniques include avoiding error-prone programming language constructs such as pointers and the use of static analysis to detect program anomalies.

11.2 Fault detection and removal -The use of verification and validation techniques that increase the chances that faults will be detected and removed before the system is used. Systematic system testing and debugging is an example of a fault-detection technique.

11.3 Fault tolerance Techniques- that ensure that faults in a system do not result in system errors or that ensure that system errors do not result in system failures. The incorporation of self-checking facilities in a system and the use of redundant system modules are examples of fault tolerance techniques.

12 Software faults cause software failures when the faulty code is executed with a set of inputs that expose the software fault. The code works properly for most inputs. Figure given below depicts a software system as a mapping of an input to an output set.

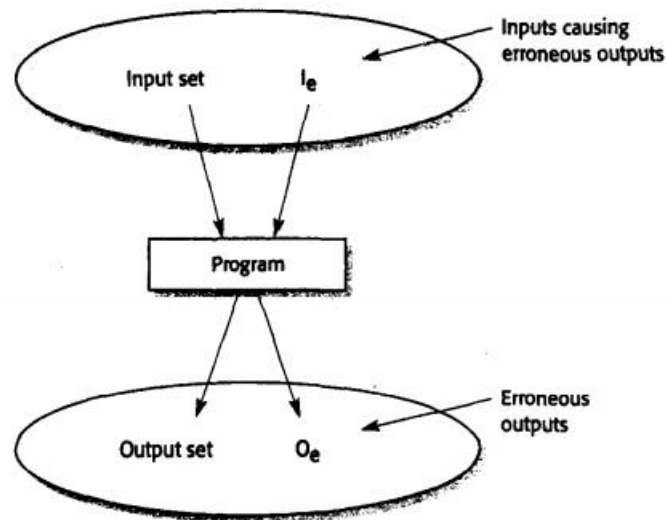


Fig 5 A system as an input/output mapping

- 13 Some of these inputs or input combinations, shown in the shaded ellipse in Figure given above, cause erroneous outputs to be generated. The software reliability is related to the probability that, in a particular execution of the program, the system input will be a member of the set of inputs, which cause an erroneous output to occur.
- 14 Each user of a system uses it in different ways. Faults that affect the reliability of the system for one user may never be revealed under someone else's mode of working as shown in figure below.

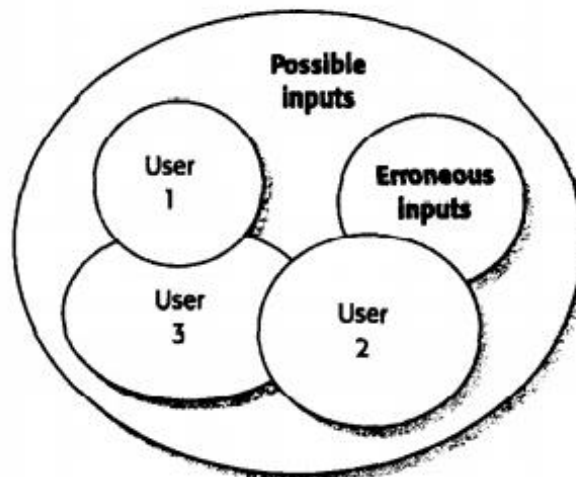


Fig 6 Software Usage Pattern

- 15 The overall reliability of a program, therefore, mostly depends on the number of inputs causing erroneous outputs during normal use of the system by most users. Software faults that occur only in exceptional situations have little effect on the system's reliability.
- 16 Users in a socio-technical system may adapt to software with known faults, and may share information about how to get around these problems. They may avoid using inputs that are known to cause problems so program failures never arise.

7.5 Safety and Security Of Software Systems

- 1 Safety-critical systems are systems where it is essential that system operation is always safe. That is, the system should never damage people or the system's environment even if the system fails. Examples of safety-critical systems are control and monitoring systems in aircraft, process control systems in chemical and pharmaceutical plants and automobile control systems.
- 2 Hardware control of safety-critical systems is simpler to implement and analyse than software control. However, we now build systems of such complexity that they cannot be controlled by hardware alone. Some software control is essential because of the need to manage large numbers of sensors and actuators with complex control laws. An example of such complexity is found in advanced, aerodynamically unstable military aircraft.
- 3 Safety-critical falls in to two classes
 - 1 Primary, safety-critical software -This is software that is embedded as a controller in a system. Malfunctioning of such software can cause a hardware malfunction, which results in human injury or environmental damage. I focus on this type of software.
 - 2 Secondary safety-critical software This is software that can indirectly result in injury. Examples of such systems are computer-aided engineering design systems whose malfunctioning might result in a design fault in the object being designed. This fault may cause injury to people if the designed system malfunctions.
- 4 System reliability and system safety are related but separate dependability attributes. Of course, a safety-critical system should be reliable in that it should conform to its specification and operate without failures.
- 5 There are several other reasons why software systems that are reliable are not necessarily safe

- 5.1 The specification may be incomplete in that it does not describe the required behaviour of the system in some critical situations. A high percentage of system malfunctions are the result of specification rather than design errors.
 - 5.2 Hardware malfunctions may cause the system to behave in an unpredictable way and may present the software with an unanticipated environment. When components are close to failure they may behave erratically and generate signals that are outside the ranges that can be handled by the software.
 - 5.3 The system operators may generate inputs that are not individually incorrect but which, in some situations, can lead to a system malfunction. The software carried out the mechanic's instruction perfectly. Unfortunately, the plane was on the ground at the time—clearly, the system should have disallowed the command unless the plane was in the air.
- 6 The key to assuring safety is to ensure either that accidents do not occur or that the consequences of an accident are minimal. This can be achieved in three complementary ways
- 6.1 Hazard avoidance the system is designed so that hazards are avoided. For example, a cutting system that requires the operator to press two separate buttons at the same time to operate the machine avoids the hazard of the operator's hands being in the blade pathway.
 - 6.2 Hazard detection and removal- The system is designed so that hazards are detected and removed before they result in an accident. For example, a chemical plant system may detect excessive pressure and open a relief valve to reduce the pressure before an explosion occurs.
 - 6.3 Damage limitation -The system may include protection features that minimise the damage that may result from an accident. For example, an aircraft engine normally includes automatic fire extinguishers. If a fire occurs, it can often be controlled before it poses a threat to the aircraft.
- 7 It is impossible to make a system 100% safe, and society has to decide whether or not the consequences of an occasional accident are worth the benefits that come from the use of advanced technologies.
- 8 Table 2 discusses different safety terminologies

Sr.No	Term	Description
1	Accident	An unplanned event or sequence of events which results In human death or Injury, damage to property or to the environment. A computer-controlled machine injuring Its operator is an example of an accident.
2	Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that detects an obstacle In front of a machine Is an example of a hazard.
3	Damage	A measure of the loss resulting from a mishap. Damage can range from m2lny people killed as a result of an accident to minor Injury or property damage.
4	Hazard Severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic where many people are killed to minor where only minor damage results.
5	Hazard Probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from probable (say 1/100 chance of a hazard occurring) to implausible (no conceivable situations are likely where the hazard could occur).
6	Risk	This is a measure of the probability that the system will cause an accident. The risk Is assessed by considering the hazard probability, the hazard severity and the probability that a hazard will result in an accident.

9 Security

9.1 Security is a system attribute that reflects the ability of the system to protect itself from external attacks that may be accidental or deliberate. Security has become increasingly important as more and more systems are connected to the Internet. Internet connections provide additional system functionality (e.g., customers may be able to access their bank accounts directly), but

Internet connection also means that the system can be attacked by people with hostile intentions.

- 9.2 Examples of attacks might be viruses, unauthorised use of system services and unauthorised modification of the system or its data. Security is important for all critical systems. Without a reasonable level of security, the availability, reliability and safety of the systems may be comprised if external attacks cause some damage to the system.
- 9.3 The reason for this is that all methods for assuring availability, reliability and safety rely on the fact that the operational system is the same as the system that was originally installed. If this installed system has been compromised in some way (for example, if the software has been modified to include a virus), then the arguments for reliability and safety that were originally made can no longer hold.
- 9.4 There are three types of damage that may be caused through external attack
 - 1 Denial of service- The system may be forced into a state where its normal service, become unavailable. This, obviously, then affects the availability of the system.
 - 2 Corruption of programs or data- The software components of the system may be altered in an unauthorised way. This may affect the system's behaviour and hence its reliability and safety. If damage is severe, the availability of the system may be affected.
 - 3 Disclosure of confidential information -The information managed by the system may be confidential, and the external attack may expose this to unauthorised people. Depending on the type of data, this could affect the safety of the system and may allow later attacks that affect the system availability or reliability.
- 9.5 There are comparable approaches that may be used to assure the security of a system
 - 1 Vulnerability avoidance -The system is designed so that vulnerabilities do not occur. For example, if a system is not connected to an external public network, then there is no possibility of an attack from members of the public.
 - 2 Attack detection and neutralisation- The system is designed to detect vulnerabilities and remove them before they result in an exposure. An example of vulnerability detection and removal is the use of a virus checker that analyses incoming files for viruses and modifies these files to remove the virus.

9.6 Table 3 discusses some security terminologies

Sr.No	Term	Description
1	Exposure	Possible loss in a computing system. This can be loss or damage to data or can be a loss of time and effort if recovery is necessary after a security breach.
2	Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.
3	Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
4	Threats	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
5	Control	A protective measure that reduces a system's vulnerability. Encryption would be an example of a control that reduced a vulnerability of a weak access control system.

Questions

Q1 Explain the four dimensions of Dependability?

Q2 What is Safety Critical Systems?

Q3 What are the three principal types of critical system? Explain the differences between these?

Q4 Suggest six reasons why dependability is important in critical systems.

Q5 Giving reasons for your answer, suggest which dependability attributes are likely to be most critical for the following systems: • An Internet server provided by an ISP with thousands of customers • A computer-controlled scalpel used in keyhole surgery • A directional control system used in a satellite launch vehicle • An Internet-based personal finance management system.

REQUIREMENT ENGINEERING PROCESS

Unit Structure

- 8.0 Objective
- 8.1 Introduction
 - 8.1.1 Requirement analysis
 - 8.1.2. Feasibility study
 - 8.1.3 Requirement Elicitation and analysis
 - 8.1.4 Eliciting and understanding stakeholder requirements.
- 8.2 Requirements Validation
 - 8.2.1 Principles of Requirements Validation.
 - 8.2.2 Requirements Validation Technique
 - 8.2.3 Requirements Management
 - 8.2.4 Requirements Management Planning
 - 8.2.5 Requirement Reviews
- 8.3 Requirement Management
 - 8.3.1 Principal stages to a change management process

8.0 Objective:

- This Chapters help you to understand.
- What is Requirement and why it is Important.
- How to manage requirement and its process.
- Different types of System Model and how it works.
- Architectural design and different component used.
- How to Structure a system and its different styles.

Requirement Engineering Processes

8.1 Introduction

Requirement engineering is a process Requirements engineering is the systematic use of proven principles, techniques, languages, and tools for the cost-effective analysis, documentation, and on-going evolution of user needs and the specification of the external behaviour of a system to satisfy those user needs.

The process to determine the requirement specification of the software is called as requirement engineering process. Both the software engineer and customer take an active role in software requirements engineering, a software engineer performs requirements analysis.

The use of Requirement Engineering Process:

If you don't analyse, it's highly likely that you'll build a very elegant software solution that solves the wrong problem.

The result is: wasted time and money, personal frustration, and unhappy customers.

Requirement engineering process includes four high level activities:

- 1) If the system is useful to the business (Feasibility study).
- 2) Discovering requirements (Elicitations and Analysis).
- 3) Converting these requirements into some standard form (Specification).
- 4) Checking that the requirements actually define the system that the customer wants (Validation).

Requirement Involved in Requirement Engineering

Depending on the type of system being developed and the specific practices of the organization(s) involved the activities or tasks that are involved in requirements engineering vary widely.

Activities involved in requirement engineering

1. Requirement inception
2. Requirement elicitation
3. Requirement analysis and negotiation
4. System modelling
5. Requirement specification
6. Requirement validation
7. Requirements management

1. Requirements Inception

Inception is a task where the requirement engineering requests a set of queries to find a software process. In requirement inception task it understands the problem and assesses with the proper solution. It cooperates with the relationship between the customer and the inventor. The overall scope and the nature of the question are decided by the developer and customer.

2. Requirements Elicitation

In requirements engineering, this is the exercise of investigating and learning the requirements of a system from stakeholders. users, customers, and other.

The requirements elicitation is also sometimes referred to as "requirement gathering".

3. Requirement Analysis and Negotiation

The tasks that are involved in requirements analysis and compromise are identification of requirements including new ones if the development is iterative and solving conflicts with stakeholders.

4. System modelling

There some engineering fields or specific study in situations in picture that need the product to be completely designed and modelled before its construction or fabrication starts. Hence, the design phase must be performed in advance.

Many fields might derive models of the system with the Lifecycle Modelling Language, whereas others might use UML.

5. Requirement specification

Requirements are documented in a formal artefact called Requirements Specification.

Nevertheless, it will become official only after validation.

A requirement specification can contain both written and graphical (models) information if necessary. Example is Software Requirements Specification (SRS).

6. Requirement validation

It is the process of checking whether the documented requirements and models are consistent and meet the needs of the stakeholder.

Only if the final draft passes the validation process, the RS becomes official.

7. Requirement Management

This phase manages all activities related to requirement after it is put into use.

The two important task of requirement engineering process are

1. Finding the requirement specifications.
2. Reviewing the requirements to ensure their correctness.

8.1.1. Requirement analysis:

Requirement analysis is a software engineering task that bridges the gap between system level requirements engineering and software design.

Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behaviour), indicate software's interface with other system elements, and establish constraints that software must meet.

The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behaviour in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints. Each of these tasks serves to describe the problem so that an overall approach or solution may be synthesized.

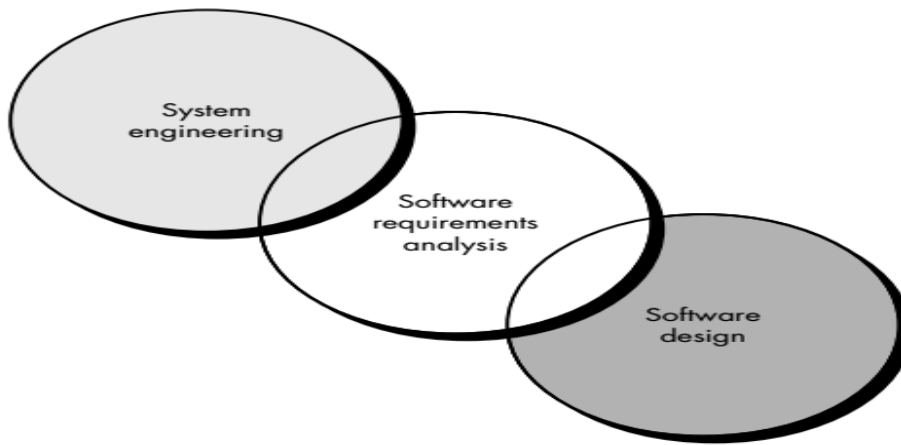


Fig: Analysis Concepts and Principles

For example, an inventory control system is required for a major supplier of auto parts. The analyst finds that problems with the current manual system include

- (1) inability to obtain the status of a component rapidly
- (2) two- or three-day turnaround to update a card file
- (3) multiple reorders to the same vendor because there is no way to associate vendors with components, and so forth.

Once problems have been identified, the analyst determines what information is to be produced by the new system and what data will be provided to the system. For instance, the customer desires a daily report that indicates what parts have been taken from inventory and how many similar parts remain.

The customer indicates that inventory clerks will log the identification number of each part as it leaves the inventory area.

All analysis methods are related by a set of operational principles:

1. The information domain of a problem must be represented and understood.
2. The functions that the software is to perform must be defined.
3. The behaviour of the software (because of external events) must be represented.
4. The models that depict information, function, and behaviour must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
5. The analysis process should move from essential information toward implementation detail.

Requirement engineering process consists of following processes:

1. Requirement elicitation and analysis
2. Requirement validation
3. Requirement management

8.1.2. Feasibility study

For all new systems the requirement engineering process should start with feasibility study.

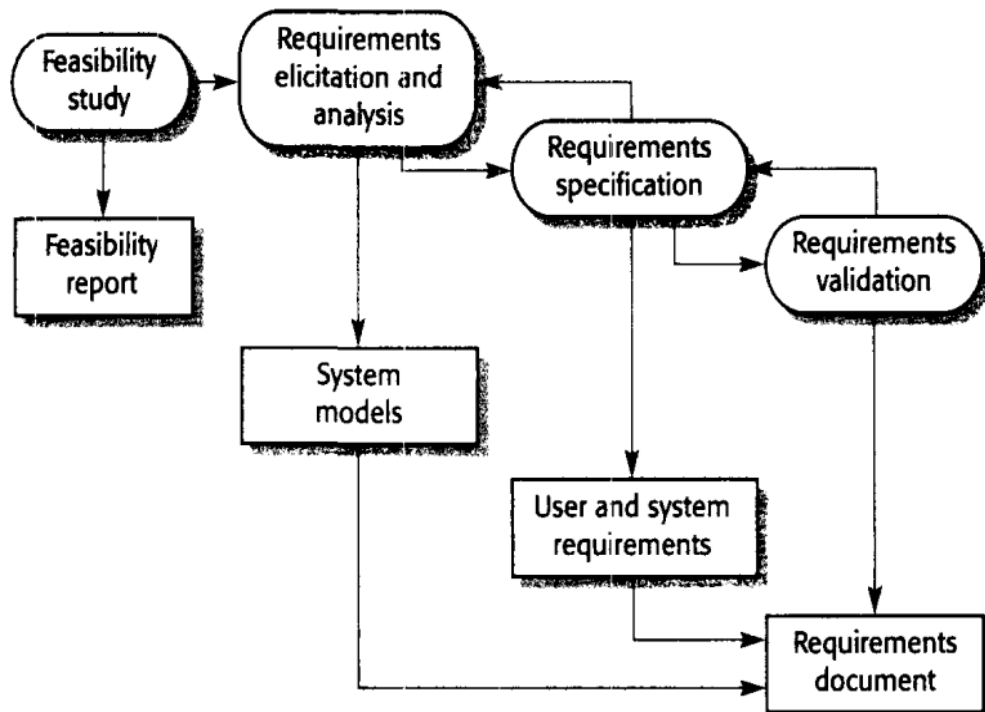


Fig. Feasibility Study

- 1) The input to the feasibility study is preliminary requirements, an outline description of how the system is intended to support business process.
- 2) The results are the report that recommends whether it is worth carrying on with the project.
- 3) Thus, a feasibility study is a short-focused study that aims to answer several questions like system contribution, system implementation and system integration.
- 4) If the system does not support the business objectives it has no real value to the business.
- 5) Carrying out a feasibility study involves information assessment, information collection and report writing.
- 6) The information assessment phase identifies the information that is required and once the information is gathered discussion with various sources can be done.

8.1.3 Requirement Elicitation and analysis

- In this activity software engineers work with customers and system end users to find about their application domain.
- This activity may involve a variety of people in the organization

- The term stakeholder is used to refer to a person or group who will be directly or indirectly affected by the system.
- It is defined as a process of requirement gathering which focuses on "What is the scope and objective of the software?" and "how it can be accomplished?".
- The users and the developers are the stakeholders who are interested in the successful development of the software.

The various tools in elicitation are meetings, interviews, questionnaire, observation collaboration, video conferencing.

The output of requirement elicitation includes the following:

- Statement of need and feasibility.
- Statement of the scope for the system.
- List of stakeholders participated in the process of requirement gathering.
- Description of the system's technical environment.
- Specification of non-functional requirements.

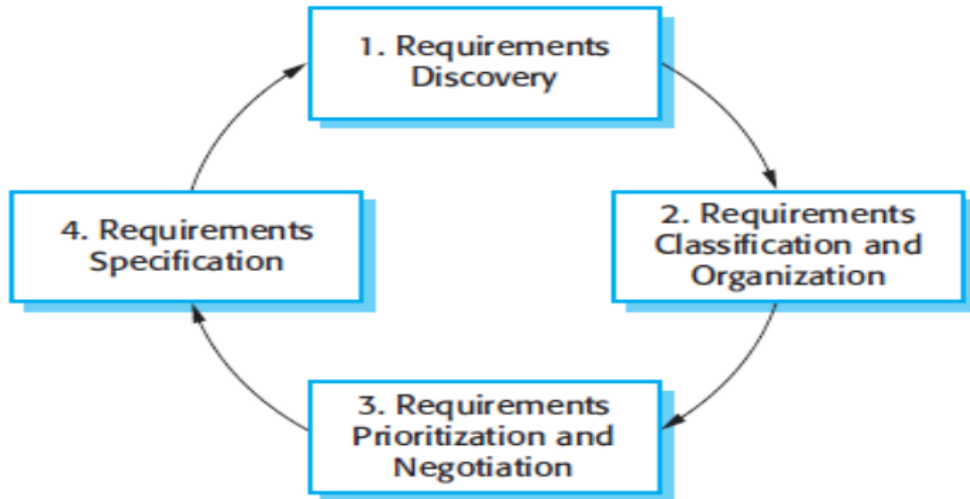
8.1.4 Eliciting and understanding stakeholder requirements is difficult because

1. Stakeholders often don't know what they want from the computer system hence they may make unrealistic demands.
2. Stakeholders naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers without experience of customer domain must understand these requirements.
3. Different stakeholders have different requirements which they may express in different ways. Requirement engineers must consider all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of the system.

The three main problems faced during the requirement gathering/elicitation process

- a) Problem of scope
- b) Problem of understanding
- c) Problem of instability

The activities used in the elicitation and analysis process are:



Requirement discovery – This is the process on interacting with stakeholders in the system to collect their requirement. Domain requirements from the stakeholders and documentation are discovered during this activity.

- It is the process of gathering information about the proposed and existing system and filtering the user requirements from it. Sources of information during the requirements discovery phase includes documentation, system stakeholders and specification of similar systems.
- **The different ways of gathering information are**
 1. **Viewpoints** – It can be used to classify stakeholders and other sources of requirements.

There are three different types of viewpoints.

- Interactor viewpoints represent people or other systems that interact directly with the system
- Indirect viewpoints represent stakeholders who do not use the system themselves but who influence the requirements in some way
- Domain viewpoints represents domain characteristics and constraints that influence the system requirements.
- Viewpoints provide different types of requirements. Interactor viewpoints provide detailed system requirements whereas indirect viewpoints are more likely to provide higher level organizational requirements and constraints and domain viewpoints normally provide domain constraints that apply to the system.

The initial identification of viewpoints that are relevant to a system can sometimes be difficult. Viewpoints that provide requirements may come from the marketing and external affairs departments.

For a non-trivial system there is huge number of viewpoints

- Interviewing – In this portion questions are put to the stakeholders about the system.
Interviews can be of open type (no predefined set of questions) or closed type (predefined set of questions).
- Interviews are good for getting an overall understanding of what the stakeholders do, how they might interact with the system and the difficulties that they face with current systems.
- Interviews are not so good for understanding the requirements from the application domain because there are subtle power and influence relationships between the stakeholders in the organization
- **Effective interviews have two characteristics**
 - They are open minded avoid perceived ideas about the requirements and are willing to listen to stakeholders
 - They prompt the interviewee to start discussion with a question
- Information from interviews supplements other information about the system from documents, user observations and so on.
- Sometimes interviews may be the only source of information but still it may miss an important thing and hence it should be used alongside other requirements techniques
- Scenarios – They are particularly useful for adding detail to an outline requirements description.
- Several forms of scenarios provide different types of information at different levels of detail about the system.
- The scenario starts with an outline of the interaction and during elicitation details are added to create a complete description of that interaction
- Scenario based elicitation can be carried out informally where the requirements engineer works with stakeholders to identify scenarios
- Scenarios may be written as text, supplemented by diagrams, screen shots etc.
- Use-cases – They are scenario-based technique for requirements elicitation.
- They have now become a fundamental feature of the UML notation for describing object-oriented system models.
- Use-cases identify the individual interactions with the system
- They can be documented with text or linked with UML models

- The UML is a de facto standard for object-oriented modelling so use cases and use case-based elicitation is increasingly used for requirements elicitation

Requirement classification and organization – This activity takes the unstructured collection of requirements and organizes them into clear clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be separate activities.

Requirement prioritization and negotiation – when multiple stake holders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders must meet to resolve differences and agree on compromise requirements.

Requirement documentation – The requirements are documented and input into next round of spiral.

Requirement Analysis:

It is a structures and organized method for identifying the set of resources to satisfy the users need.

It acts as a transformation between the system need and the design concept.

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

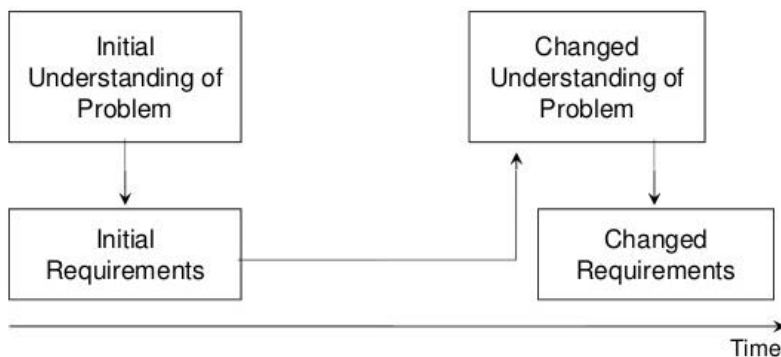
8.2 Requirement Validation

- It is concerned with showing that the requirements define the system that the customer wants.
- It overlaps analysis in that it is concerned with finding problems with the requirements.
- It is important because errors in requirements documentation can lead to extensive rework costs when they are discovered during development process.
- The cost of fixing a requirements problem is much greater than repairing design or coding errors
- **Various checks carried out on requirements in requirements document are**
 1. **Validity checks** A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with

different needs and any set of requirements is inevitably a compromise across the stake holder community

2. **Consistency checks** Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. **Completeness checks** the requirements document should include requirements that define all functions and the constraints intended by the system user.
4. **Realism checks** Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.
5. **Verifiability** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Requirements evolution



There are several requirements validation techniques that can be used individually or in conjunction with one another:

1. **Requirements reviews** the requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.
2. **Prototyping** In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
3. **Test-case generation** Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this

usually means that the requirements will be difficult to implement and should be re-considered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

8.2.1 Principles of Requirement Validation.

- 1 Involvement of the correct stakeholders.
- 2 Unravelling the identification and the correction of errors.
- 3 Validation from different views.
- 4 Satisfactory change of documentation type.
- 5 Construction of development artifacts.
- 6 Recurring authentication (Validation).

8.2.2 Requirements Validation Technique

- **Requirements reviews** – The requirements are analysed systematically by a team of reviewers
- **Prototyping** – In this approach to validation an executable model of the system is demonstrated to end users and customers
- **Test case generation** – Tests for the requirements are devised as a part of validation process. Developing tests from the user requirements before any code is written is an integral part of extreme programming
- Requirement review is a manual process that involves people from both client and contractor organizations.
- They check the requirements document for irregularities and errors.
- Requirements reviews can be informal (involves contractors discussing requirements with stake holders) and formal (the development team walks through the system requirements)
- Reviewers may also check for
 - Verifiability
 - Comprehensibility
 - Traceability
 - Adaptability

8.2.3 Requirements Management

Requirements for the large system keeps on changing due to which stakeholders understanding of problem is constantly changing. It is hard for the users and system customers to anticipate what effects the new system will have on organization.

Requirement management is the process the principal concerns are of managing the changes to requirements.

- Managing the relationships between requirements.
- Managing priorities between requirements.

- Managing the dependencies between different documents, specification and other documents produced in the systems engineering process.
- Managing changes to decided requirements.

New needs and priorities are discovered

- Large systems have different users having different requirements and priorities.
- After delivery new features may be added for user support if the system is to meet its goal.
- Business and technical environment changes after installation and these changes must be reflected in the system.

8.2.4 Requirements Management Planning

- Planning is an essential first stage in the requirements management process.
- During the requirement management stage, following are decided: -
 1. **Requirements reviews** the requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.
 2. **Prototyping** In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
 3. **Test-case generation** Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be re-considered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

Requirements management needs automated support and the software tools, and this should be chosen during the planning phase.

The needed tool support is: -

- 1) **Requirement storage:** - The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
- 2) **Change management:** - The process of change management is simplified if active tool support is available.
- 3) **Traceability management:** - some tools are available which use natural language processing techniques to help discover possible relationship between requirements.

8.2.5 Requirement Reviews

The various techniques used for requirement validations are

- a) Requirement reviews
- b) Perspective-based reading
- c) Validation through prototypes
- d) Using checklists for validation

Requirement review is a technique used to validate the requirements by a group of people. It is a formal process which involves readers from the both sides of clients and developers. Reviews help customers and developers to resolve problems at early stages of SDLC.

Requirements review process consists of following points

- a) **Plan Review:** Team is selected, time and place are decided for the review meeting with all the requirements to be checked.
- b) **Distribute Documents:** Brochures are distributed among the review team members so, each member gives their review on the documentation.
- c) **Prepare for Review:** Each Persons read the relevant documents for inconsistencies, conflicts, omissions, and other problems before review meeting.
- d) **Hold Review Meeting:** Individual remarks and glitches are discussed, set of actions to address the problem is agreed.
- e) **Follow-up Actions:** Checks for the work completion and the progresses of the task given to the individual person of team.
- f) **Rescript Document:** Documents were checked by Team members for rechecking or reviewing purpose

Requirements review process

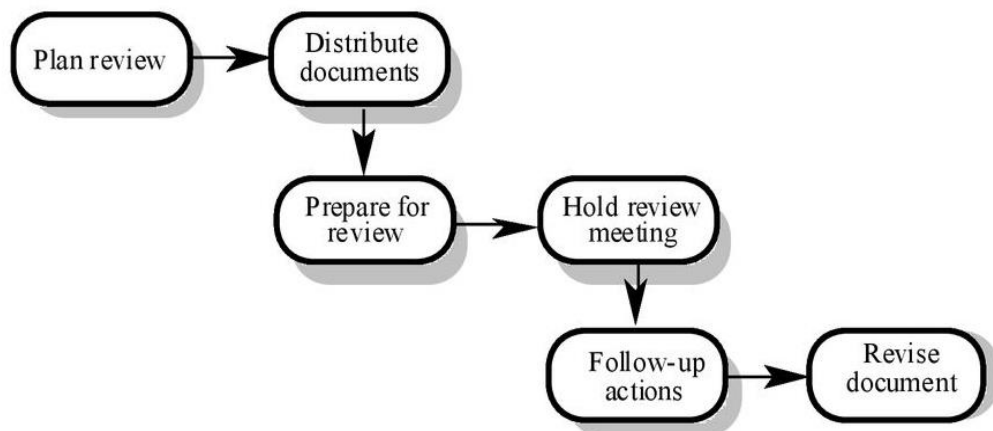


Fig: Requirement Review Process

8.3 Requirement change Management



Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation.

The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

It begins from the changes being made in the environment in which finished product is considered to be used, business changes, regulation changes, errors in the original definition of requirements, limitations in the technology, and changes in security environment and so on.

The activities that are included in requirements change management are receiving change requests from the stakeholders, recording the received change requests, analysing and determining the desirability and process of implementation, implementation of change request, and quality assurance for the implementation and closing the change request.

After this the data of the change requests are compiled, analysed and appropriate metrics are derived then fit into the organizational knowledge repository.

8.3.1 Principal stages to a change management process:

1. Problem analysis and change specification: -

During this stage, the problem or the change proposal is analysed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal or decide to withdraw the request.

2. Change analysis and costing: -

The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether to proceed with the requirements change.

3. Change implementation:

The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization.

Graded Questions:

1. What are the underlying principles that guide analysis work?
2. Explain Analysis Concept and Principles with neat diagram.
3. What is requirement Management? Explain.
4. Explain Requirements Validation Technique
5. What are different Principal stages to a change management process?

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach, Roger Pressman , Tata Mcgraw-hill , Seventh

SYSTEM MODELS

Unit Structure

- 9.0 Objective
- 9.1 Introduction to System Models
 - 9.1.1 Essential element of system model
 - 9.1.2 Features of Modelling Techniques
- 9.2 Type of System Model
 - 9.2.2 Context Models
 - 9.2.2 Behavioural Model
 - 9.2.2.1 Data Flow Models
 - 9.2.2.2 State Machine Models
- 9.3 Data Models
- 9.4 Object Model
- 9.5 Structured Method
- 9.6 Interaction Model
- 9.7 Use case Modelling
- 9.8 Sequence diagram

9.0 Objective

In this chapter you will understand how system were created and what are the different element, what are the different system model and how and where this model is useful

9.1 Introduction to System Models

Every computer-based system can be modelled as an information transform using an input-processing-output template.

Using a representation of input, processing, output, user interface processing, and self-test processing, a system engineer can create a model of system components that sets a foundation for later steps in each of the engineering disciplines.

The system engineer allocates system elements to each of five processing regions within the template:

- (1) user interface
- (2) input
- (3) system function and control
- (4) output, and (5) maintenance and self-test.

Like nearly all modelling techniques used in system and software engineering, the system model template enables the analyst to create a hierarchy of detail.

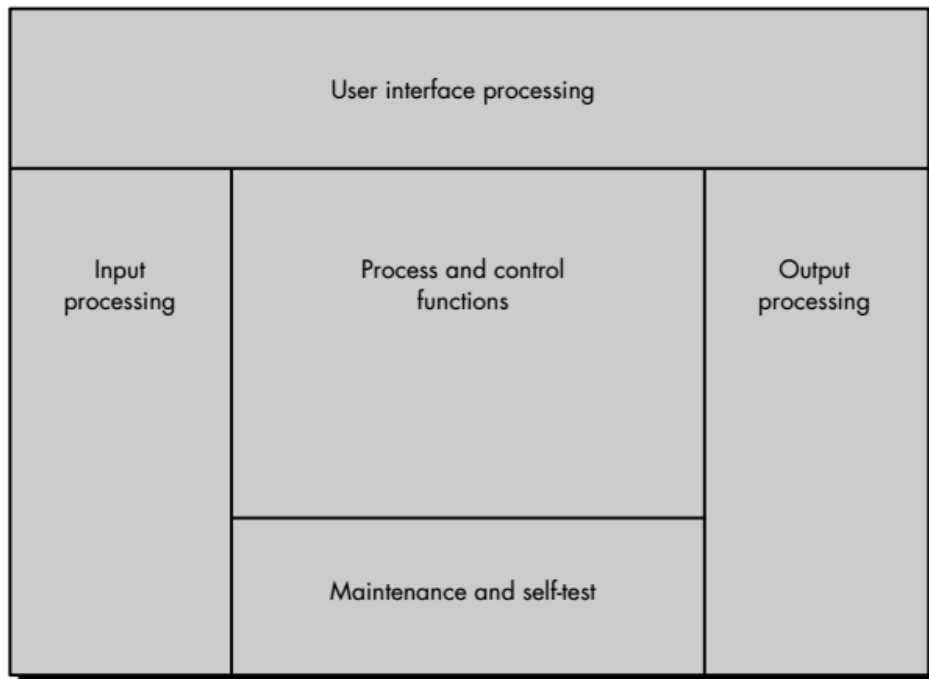


Fig: System Modelling Template

System modelling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

System modelling uses graphical notation, Unified Modelling Language (UML). Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineer implementing the system and after implementation to document the system's structure and operation.

You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.

2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.

The UML has many diagram types and so supports the creation of many different types of system model.

9.1.1 The three essential elements of system models are

a) Environmental Model:

It defines the scope of the proposed system and its boundaries. It consists of statement of purpose, context diagram and events of the system

b) Behavioural Model:

It describes the functional requirements, internal behaviour and data entities of the system. It consists of ER diagram, DFD, State Transition diagram

c) Implementation Model:

It describes the design specification of the software and consist of software architecture, data design, interface design and component design

d) Structural Model:

It emphasises on modelling the structure of the data that is processed by the system.

9.1.2 Features of Modelling Techniques

1. Easy to use: Can use the model and understand the model working and can give proper inputs and receive the outputs from the system.
2. Contain few about powerful modelling objects to enable easy learning: Structure of the model must be easy to understand and handle properly, so if any error occur end user can perform respective task to remove or avoided the errors
3. Help to handle complexity of the system: Can able to work on all the modules or phases of the system, so can receive the proper output from the system without making any error or mistake during the function of the system.

9.2 Type of System Model

Model are of following types:

- **Data processing Model:** Data processing model showing which component and system are used to processes the data and how the data processed at different stages.

- **Composition Model:** Composition model display where the interface and connection is between the entities and how entities are calm other entities.
- **Architectural Model:** Architectural model shows what are the main components of the system and main sub-systems in the project model.
- **Classification Model:** Classification model showing how entities are interrelated to each other and have common characteristics.
- **Response Model:** Response model showing the system's reaction to events.

UML five diagram types could represent the essentials of a system:

1. Activity diagrams, which show the activities involved in a process or in data processing.
2. Use case diagrams, which show the interactions between a system and its environment.
3. Sequence diagrams, which show interactions between actors and the system and between system components.
4. Class diagrams, which show the object classes in the system and the associations between these classes.
5. State diagrams, which show how the system reacts to internal and external events.

9.2.1 Context Models

- At an early stage in the requirements elicitation and analysis process boundaries of the system must be decided involving system stakeholders
- In some cases, the boundary between a system and its environment is relatively clear.

For example, where an automated system is replacing an existing manual or computerized system the environment of the new system is usually the same as the existing system.

whereas in other cases the stakeholders decide the boundary.

- For example, in the library system the user decides the boundary whether to include library catalogues for accessing or not
- Once some decisions on the boundaries of the system have been made part of the activity is definition of that context. Producing a simple architectural model is the first activity.

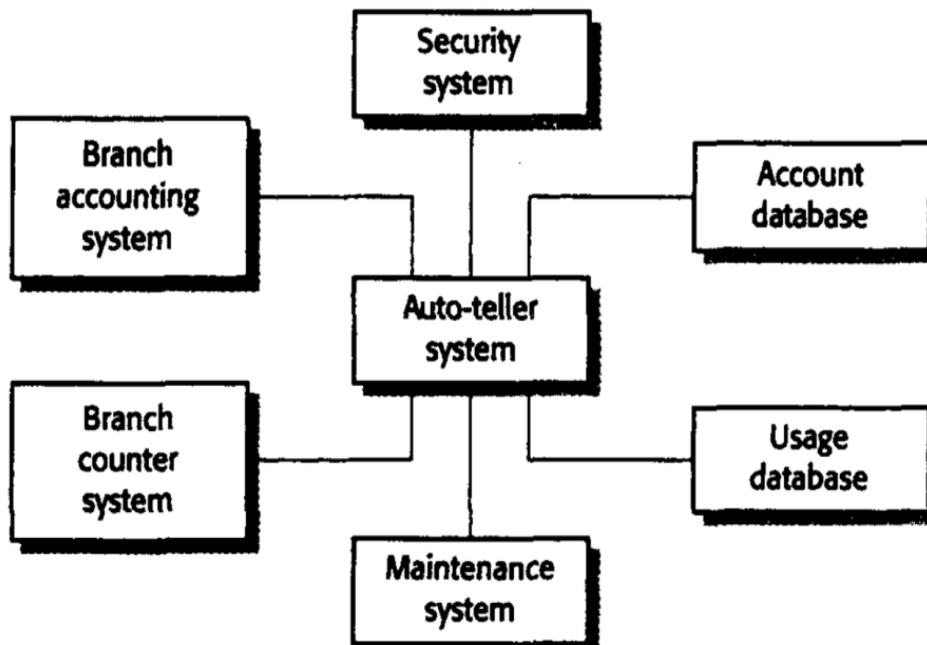


Fig: Context Model for ATM

In the above figure each ATM is connected to account database, local branch accounting system, a security system, and a system to support machine maintenance.

- The system is also connected to usage database that monitors how the network of ATM is used and to a local branch counter system.
- This counter system provides services such as backup and printing.
- These therefore need not be included in the ATM system itself.
- Architectural models describe the environment of the system but do not show the relationships between the other systems in the environment and the system that is being specified.
- Simple architecture models are supplemented by other models such as process models that show the process activities by the system.

9.2.2 Behavioural Model

- Used to describe the overall behaviour of the system
- These are of two types
 1. Data Flow Models which model the data processing in the system
 2. State Machine Models which model how the system reacts to events
- Some systems are driven by data, so data flow model is enough to represent their behaviour.
- Real time systems are often driven with minimal data processing and hence state machine model is most effective.

9.2.2.1 Data Flow Models

- An intuitive way of showing how data is processed by a system
- At the analysis level they should be used to model the way in which data is processed in the existing system
- Notations used in this model represents functional processing (rounded rectangles) data stores (rectangles) and data movements between functions (labelled arrows)
- Used to show how data flows through a sequence of processing steps
- The data is transformed or processed before moving to the next step which are known as software processes or functions
- Data flow models are valuable because tracking and documenting how the data associated with a particular process moves through the system.
- They are simple and intuitive and is usually possible to explain them to potential system users.

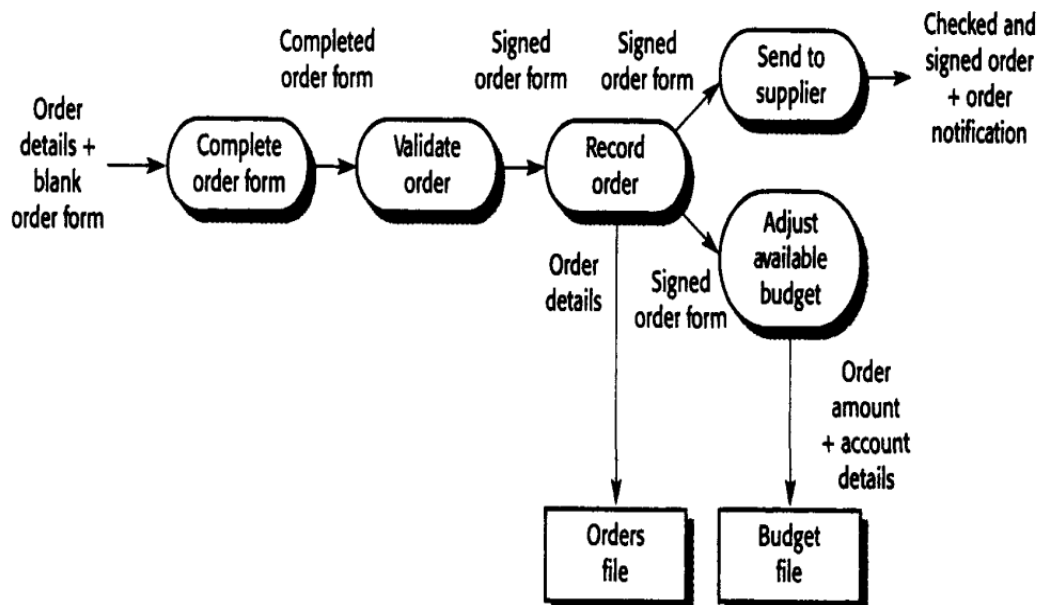


Fig: Data Flow Model for Order Processing

9.2.2.2 State Machine Models

- It describes how a system responds to internal or external events and shows the system states and events that cause transitions from one state to another but does not show the flow of data within the system
- This type of model is often used for modelling real time systems

- These models are an integral part of real time design methods and assumes that at any time the system is in one of the numbers of possible states
- The problem with the state machine approach is that the number of possible states increases rapidly and therefore some structuring is needed.

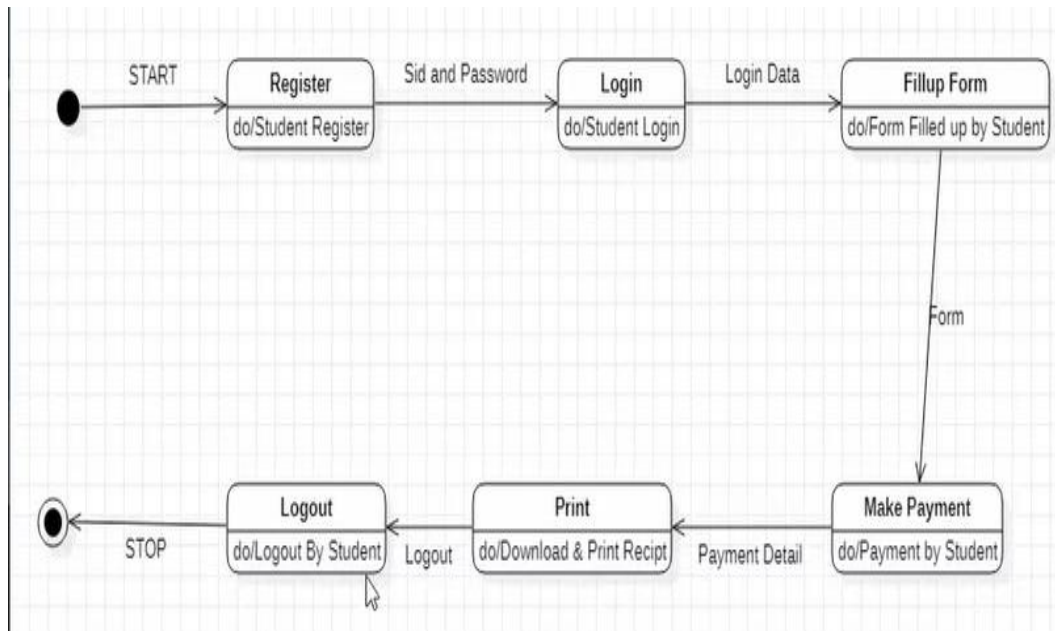


Fig: Form Filling and Payment by Student

In the Given figure we can see that the start state and stop state, start state begin with the registration of the student after registration student will receive his student ID and password, using which he will log in into a system where he will fill up the admission form, for which he had to pay the payment accordingly as done with the payment.

Then need to print the payment slip and field form ask at work is done he can log out from the system and the state will be stopped state machine is nothing, but which shows us of progress in the system.

This diagram can be drawn using star UML software.

9.3 Data Models

1. An important part of system modelling is defining the logical form of data processed by the system. These are called as semantic data models.
2. The most widely used data modelling technique is ERA (Entity-Relation-Attribute) modelling.
3. The relationship models devised from this system are in 3NF and hence they been widely used.

4. Because of the explicit typing and the recognition of the subtypes and super types it is also straight forward to implement these models using object-oriented databases.
 5. Data models lack detail, and more descriptions of ERA must be maintained by using data dictionaries.
 6. Data dictionaries are used to develop system models.
 7. It is simply an alphabetical list of names included in model.
 8. The advantages of data dictionary are it checks for the uniqueness and warns against name duplications, and it stores all data in a single place.
 9. Data model is dependent on data, data relationship, data semantics and data constraints.
 10. data model provides the various details such as information to be stored and is of primary use when the final product is created.
 11. A data model determines the structure of data explicitly. In the data modeling graphical notation are used and the data model are also specified.
 12. Data model organizes element of data and standardize how they relate to one another to the property of real-world entity.
 13. Sometime data model can be a formalization of the object and relationship found in a particular application domain for instance the customer, process products and older found in manufacturing organization.
9. The following figure is an example of data model.

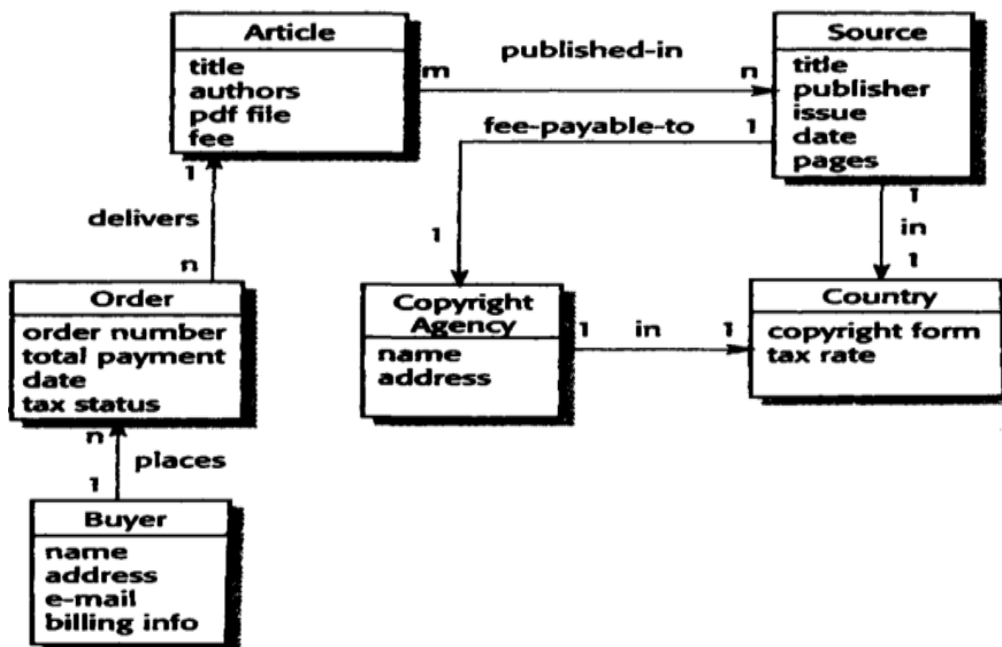


Fig: Semantic Data Model

9.4 Object Model

- Expressing the system requirements using object model, designing using objects and developing using languages like C++ and Java.
- Object models developed during requirements analysis are used to represent both data and its process. They combine some uses of dataflow and semantic models.
- They are also useful for showing how entities in the system may be classified and composed of other entities. Objects are executable entities with attributes and services of the object class and many objects can be created from a class.
- The following diagram shows an object class in UML as a vertically oriented rectangle with three sections – name of the object, class attributes, operations associated with the object.

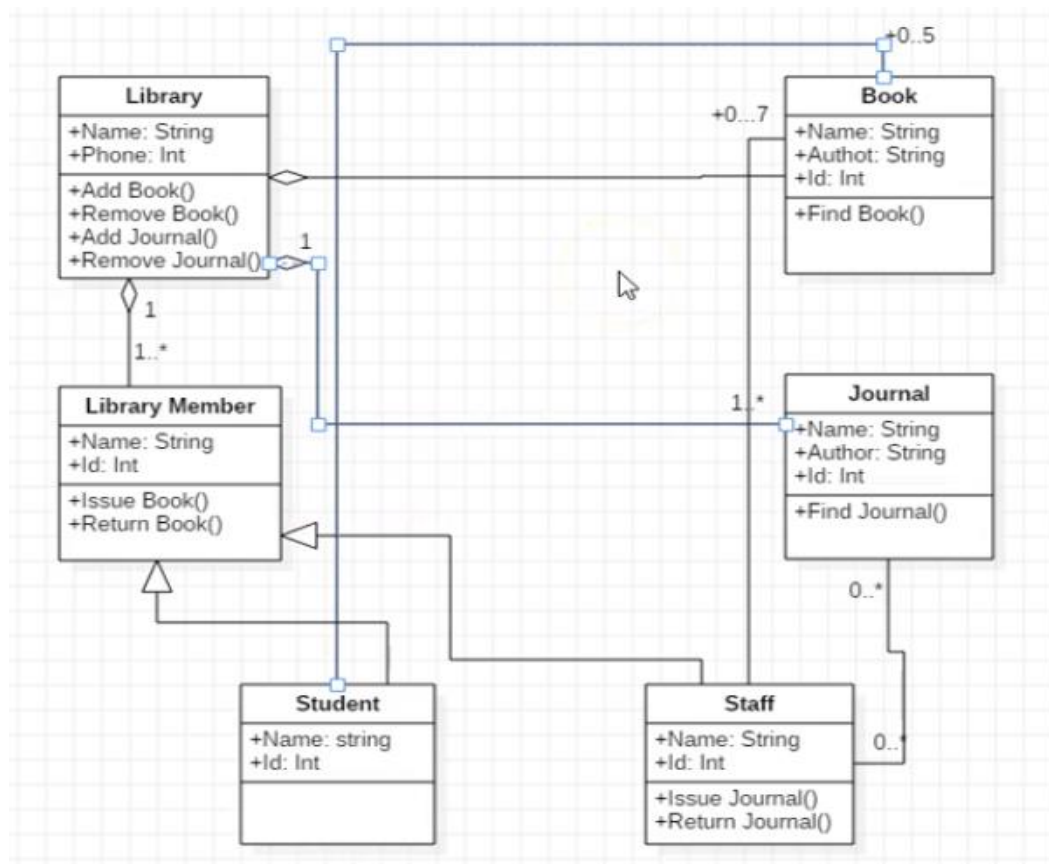


Fig: Library Management System

In the about library management system, we can have multiple objects with their attributes and operations for example we have a library class which include attribute like name and phone an operation like add book remove book add Journal remove Journal which help us to provide oral information about that class.

This object model we can call it as inheritance model as well in object-oriented modelling it is imperative to identify the class off object that are important in domain that is Bing studied.

The class of the object are then organized into taxonomies, and it is nothing but a classification scheme that shows how an object class is related to other class through common attribute and service.

The object class are capable to inherit their attribute and service from one or more yeah super class you can see in the example where library members he's a super class and student and staff are the subclass which inherited the attribute of superclass.

9.5 Structured Methods

- Systematic way of producing models of an existing system or proposed system
- Provide a framework for detailed system modelling as part of requirements elicitation and analysis
- Have their own preferred set of system models and usually define a process that are used to derive these models and set of rules and guidelines that apply to the models
- CASE tools usually used support model editing, coding, report generation and some model checking capabilities
- Have been applied in many large projects because they use standard notations and ensure standard design documentation.
- Suffer from following weakness
 - Do not provide effective support for understanding non-functional requirements
 - Do not include guidelines whether a method is appropriate, nor do they advise on how they can be adapted for a particular project.
 - Produce too much documentation.

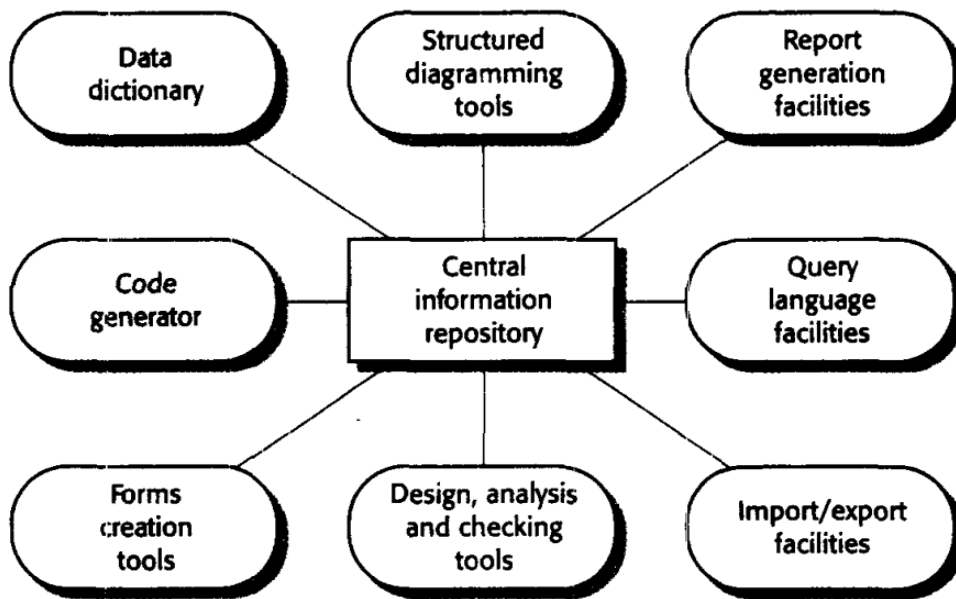


Fig: Central Information Repository

Diagram Editors

These editors are used to create object models, data models, and behavioural models and so on, they are not just drawing tools but are aware of the types of entities in the diagram.

The editors are capable of capturing information about these entities and save this information in the central repository.

Design Analysis and checking tools

These tools process the design and report on errors and anomalies.

They may be integrated with the editing system so that user errors are trapped at an early stage in the process.

Repository Query Languages

These languages facilitate the designers determine designs to and associated design information in the repository.

Data Dictionary

It has responsibility of maintaining information about the entities used in a system design.

Report Definition and generation tools

These tools accept information from the central store and automatically generate system documentation.

Forms definition tools

These tools facilitate screen and document formats to be specified.

Import/Export Facilities

These facilities allow the interchange of information the central repository with other development tools

Code generators

The code generator are responsible for generating code or code skeletons automatically from the design captured in the central store.

9.6 Interaction Model

All systems involve interaction of some kind.

This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system.

Interaction Model helps in, Modelling user interaction is important as it helps to identify user requirements.

Modelling system to system interaction highlights the communication problems that may arise.

Modelling component interaction helps us understand if a proposed system structure.

There are two related approaches to interaction modelling:

1. Use case modelling, which is mostly used to model interactions between a system and external actors (users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case models and sequence diagrams present interaction at different levels of detail and so may be used together.

The details of the interactions involved in a high-level use case may be documented in a sequence diagram.

The UML also includes communication diagrams that can be used to model interactions.

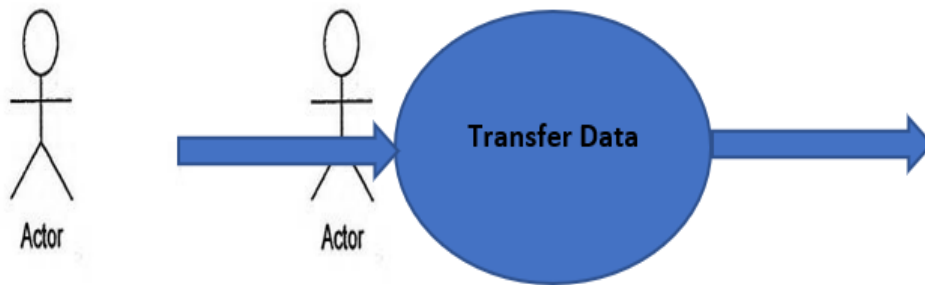
We won't discuss these here as they are an alternative representation of sequence charts. In fact, some tools can generate a communication diagram from a sequence diagram.

9.7 Use case Modelling

use case modelling is widely used to support requirements elicitation. A use case can be taken as a simple scenario that describes what a user expects from a system.

Each use case represents a discrete task that involves external interaction with a system.

In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures.



Medical Receptionist Patient Record System

Notice that there are two actors in this use case: the operator who is transferring the data and the patient record system. The stick figure notation was originally developed to cover human interaction, but it is also now used to represent other external systems and hardware.

Use case diagrams give a simple overview of an interaction so you have to provide more detail to understand what is involved.

Can use case modeling the actor there are two types of actor when who initiate and another who will react to the system there must be at least one interaction with the use cases with actors the initiator always keep left side of the system and reaction actor always keep right side of the system.

The use cases in the system must be arrange in proper logical order only so that actor can have a proper interaction with use cases within the system

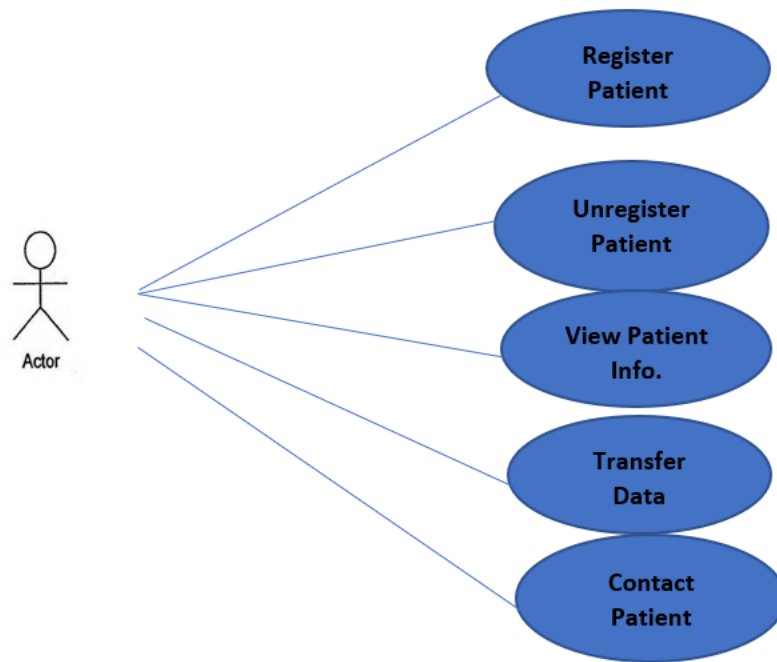


Fig: Use case involving the role ‘medical receptionist’

9.8 Sequence diagram

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves.

The UML has a rich syntax for sequence diagrams, which allows many kinds of interaction to be modelled. I don't have space to cover all possibilities here, so I focus on the basics of this diagram type.

As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.

Below Figure is an example of a sequence diagram that illustrates the basics of the notation.

This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information

The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.

Interactions between objects are indicated by annotated arrows.

The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation).

You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values. In this example, we can see the notation used to denote alternatives.

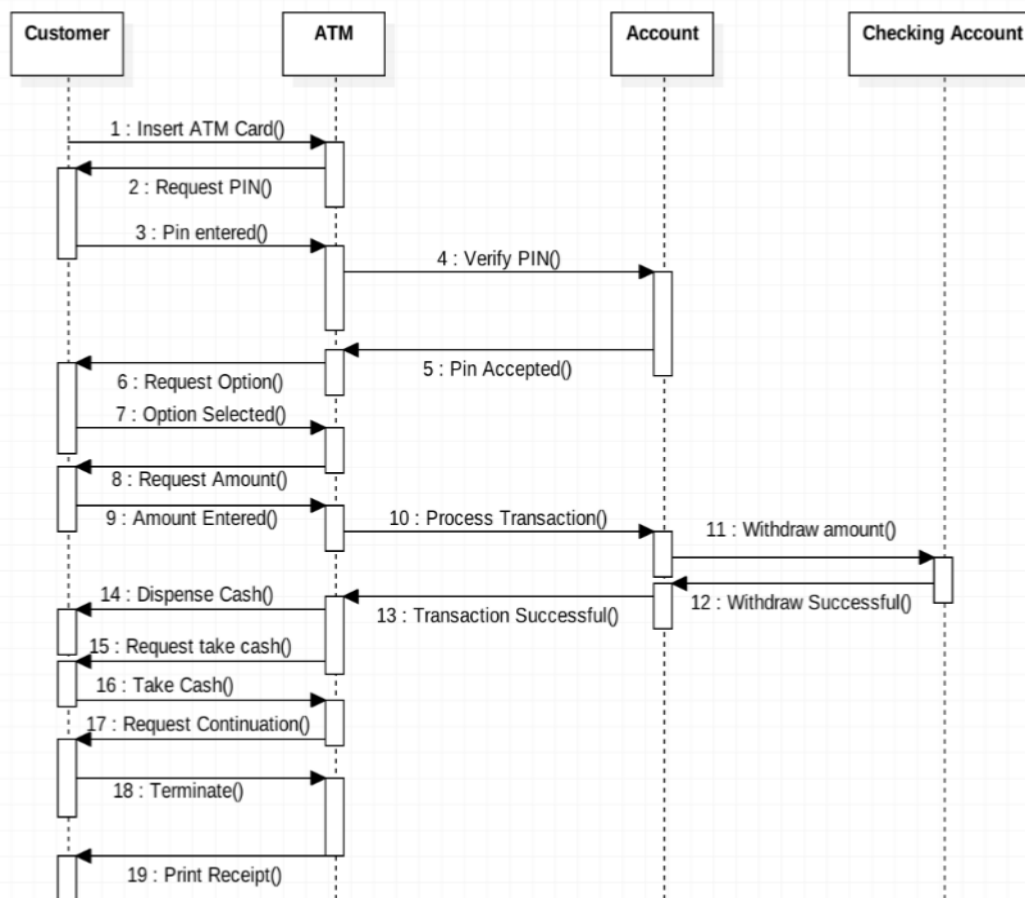


Fig: Sequence Diagram

In the given sequence diagram, we can see the sequence of steps to complete the process in ATM system.

Graded Question

1. What are the advantages of Architectural Design? Which factors are dependable during the design?
2. What is the design perspective of architectural design? Explain.
3. What are the three system organization styles of architectural design? Explain in brief.
4. Explain use case diagram with neat diagram.
5. Write a short note on object model.

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

ARCHITECTURAL DESIGN

Unit Structure

- 10.1 Introduction to Architectural design
 - 10.1.1 Architectural Design
 - 10.1.2 Software architectures design levels
- 10.2 Architectural design decisions
 - 10.2.1 The various attributes of architecture
- 10.3 Architectural View
 - 10.3.1 Architectural Design Processes
- 10.4 System Organization
 - 10.4.1 Shared data repository style
 - 10.4.2 A shared service and server's style (Client-Server Model)
 - 10.4.3 Pipe and Filter architecture
 - 10.4.4 An abstract machine or layered style.
- 10.5 Modular Decomposition Styles
 - 10.5.1 Object -oriented decomposition:
 - 10.5.2 Invoice processing system
 - 10.5.3 Function Oriented Pipelining or Data flow model
- 10.6 Reference architectures
- 10.7 Application architecture
 - 10.7.1 As a software designer
 - 10.7.2 Type of allocation system

Objective:

1. Understand the concept of software architecture
2. Understanding the design
3. Know the architectural patterns

10.1 Introduction to Architectural design

Architectural design is concerned with understanding how a system should be organised and design overall structure of system. In the model of software development process architectural design is the first stage in software design process.

Application systems are intended to meet a business or organizational need. All businesses have much in common—they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector specific applications.

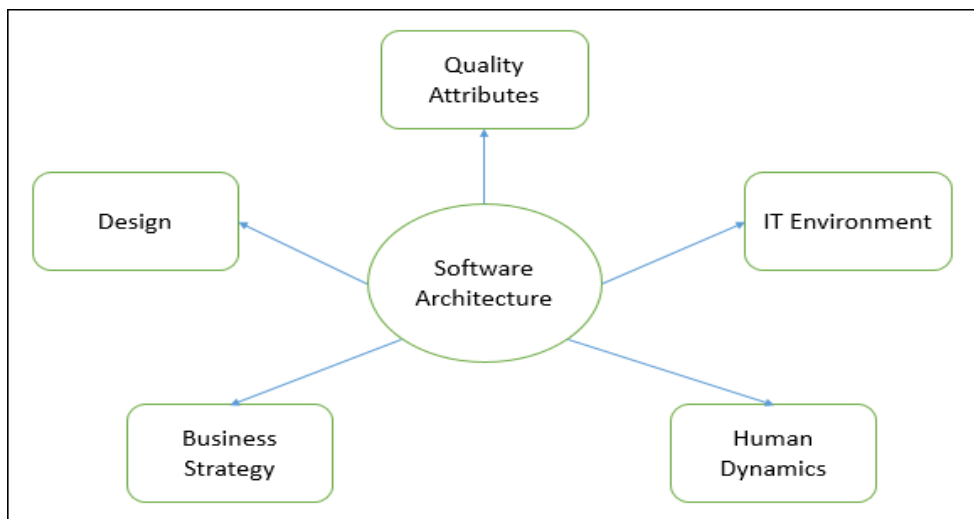
Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type.

The application architect may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re implementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

10.1.1 Architectural Design

The architecture of a system describes its major components, their relationships structures, and how they interact with each other. Architectural design is also called as high-level design.

Software architecture and design includes several related factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In Architecture, non-functional

decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

10.1.2 Software architectures can be designed at two levels of concept:

- Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.

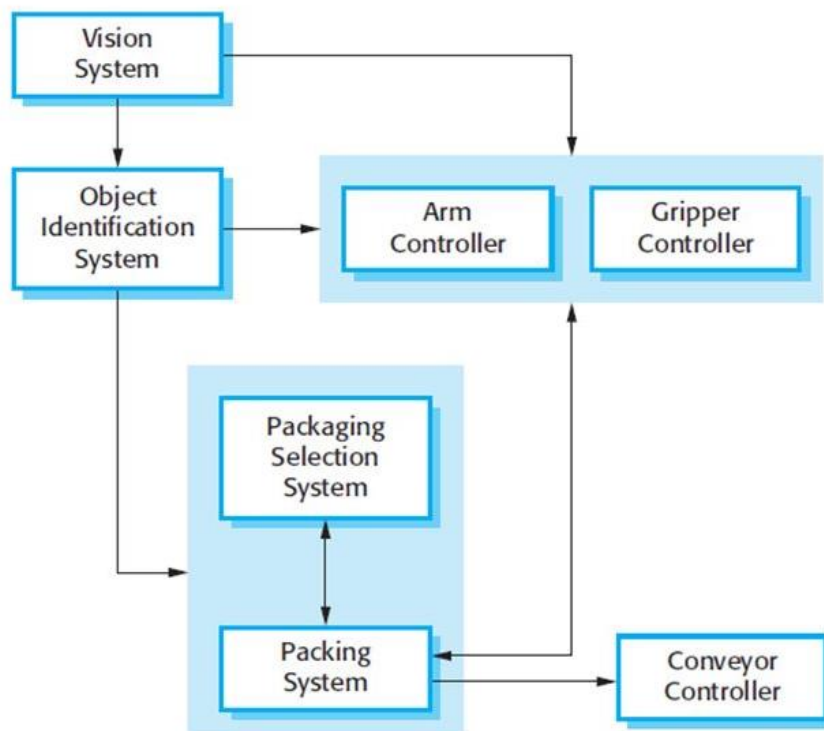


Fig: The architecture of a packing robot control system

There are Three advantages of explicitly designing and documenting software architecture:

1. **Stakeholder** communication the architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. **System analysis** Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, reliability, and maintainability.

- 3. Large-scale reuse** A model of a system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 16, it may be possible to develop product-line architectures where the same architecture is reused across a range of related systems.

The apparent contradictions between practice and architectural theory are because there are two ways in which an architectural model of a program is used:

1. As a way of facilitating discussion about the system design A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.
2. As a way of documenting an architecture that has been designed The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

10.2 Architectural design decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?
2. How will the system be distributed across a number of cores or processors?
3. What architectural patterns or styles might be used?
4. What will be the fundamental approach used to structure the system?
5. How will the structural components in the system be decomposed into subcomponents?
6. What strategy will be used to control the operation of the components in the system?
7. What architectural organization is best for delivering the non-functional requirements of the system?
8. How will the architectural design be evaluated?
9. How should the architecture of the system be documented?

Because of the close relationship between non-functional requirements and software architecture, the architectural style and structure that you choose for a system should depend on the non-functional system requirements:

10.2.1 The various attributes of architecture are as follows:

1. Performance

If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network.

This may mean using a few relatively large components rather than small, fine-grain components, which reduces the number of component communications. You may also consider run-time system organizations that allow the system to be replicated and executed on different processors.

2. Security

If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.

3. Safety

If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.

4. Availability

If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe two fault-tolerant system architectures for high-availability systems.

- 5. Maintainability** If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed.

10.3 Architectural View

There are different opinions as to what view are required, some of the view that are suggested.

1. A logical view, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.
2. A process view, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
3. A development view, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
4. A physical view, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

10.3.1 Architectural Design Processes are as follows

1. System Structuring
2. Control Modelling
3. Modular Decomposition

10.4 SYSTEM ORGANIZATION

System Organization reflects the basic strategies that is used to structure a system.

Three organisational styles are widely used:

- 9.4.1 Shared data repository style:** Sub-System must exchange the data.

- Shared data is held in a central database
- Each sub-system maintains its own database and pass it to other sub-system

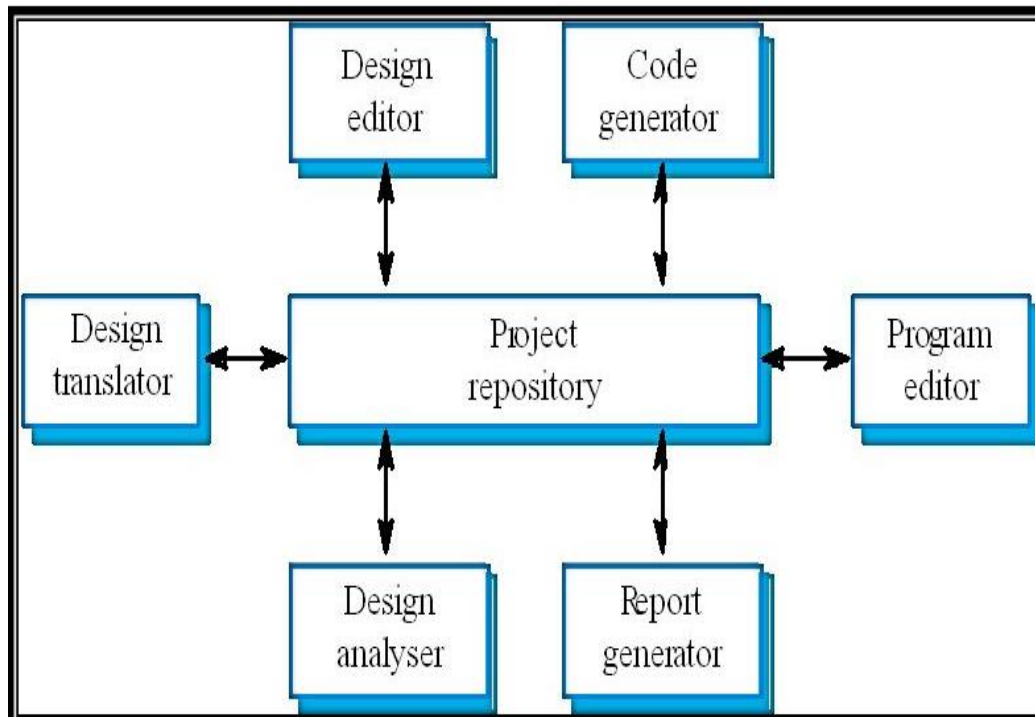


Fig: A repository architecture for an IDE

9.4.2 A shared service and server's style (Client-Server Model)

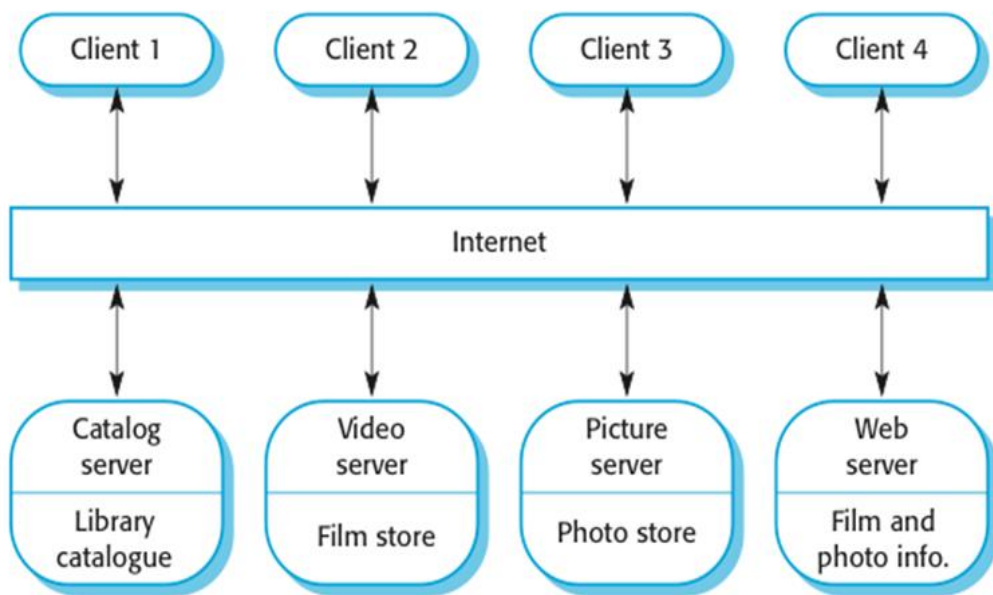
It is a distributed system model which shows how data and processing is distributed across a range of components.

A system that follows the client-server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server, which offers programming language compilation services.
2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
3. A network that allows the clients to access these services. Most client-server systems are implemented as distributed systems, connected using Internet protocols.

Client-server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of

the system. Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a Server through remote procedure calls using a request-reply protocol such as the http protocol used in WWW.



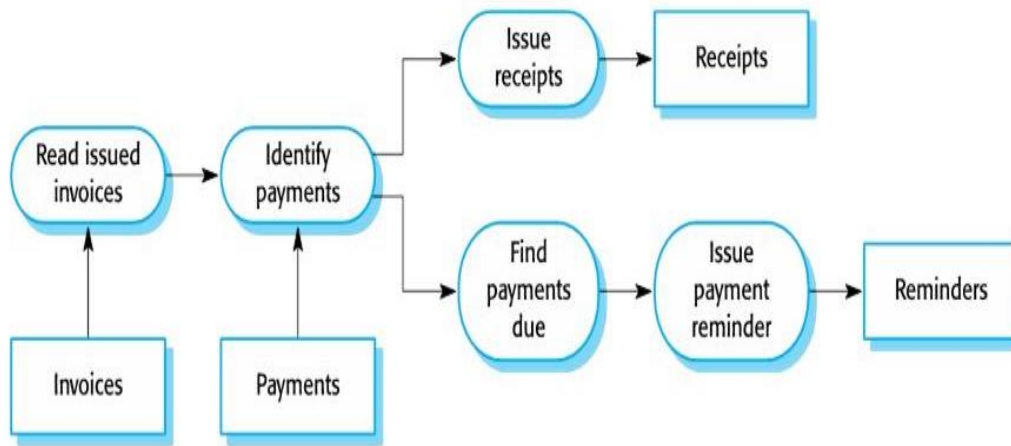
10.4.3 Pipe and Filter architecture

This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform, Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

The name 'pipe and filter' come from the original Unix system where it was possible to link processes using 'pipes'. These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term

'filter' is used because a transformation filters out the data it can process from its input data stream.

Variants of this pattern have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data processing systems (e.g., a billing system). The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently.



An example of this type of system architecture, used in a batch processing application, is shown in above Figure. An organization has issued invoices to customers.

Once a week, payments that have been made are reconciled with the invoices.

For those invoices that have been paid; a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued. Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed.

Although simple textual input and output can be modelled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections.

It is difficult to translate this into a form compatible with the pipelining model.

10.4.4 An abstract machine or layered style.

It is used to organise the system systematically into layers.

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. separates elements of a system,

allowing them to change independently. For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model. The layered architecture pattern is another way of achieving separation and independence.

This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer.

As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations of an application system. Only the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or database.

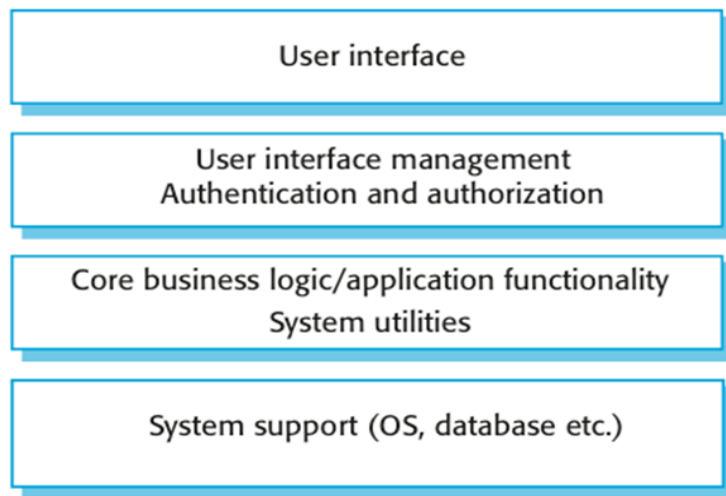


Fig: A generic layered architecture

Above Figure is an example of a layered architecture with four layers.

The lowest layer includes system support software-typically database and operating system support.

The next layer is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components.

The third layer is concerned with user interface management and providing user authentication and authorization, with the top layer providing user interface facilities.

Of course, the number of layers is arbitrary. Any of the layers in Figure could be split into two or more layers.

10.5 Modular Decomposition Styles

Two modular decomposition models covered

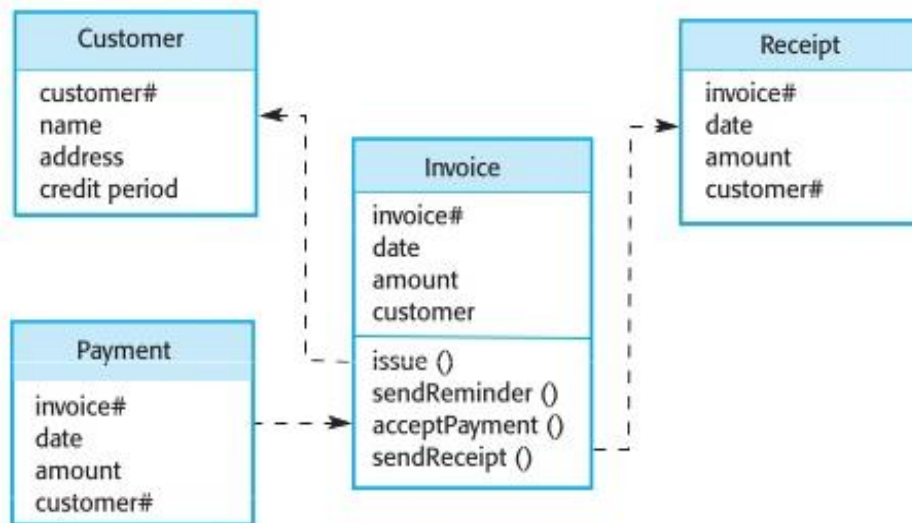
1. An object model where the system is decomposed into interacting objects
2. A data -flow model where the system is decomposed into functional modules which transform inputs to outputs. Also known as the pipeline model.

If possible, decisions about concurrency should be delayed until modules are implemented

10.5.1 Object -oriented decomposition:

1. Structure the system into a set of loosely coupled objects with well -defined interfaces
2. Object-oriented decomposition is concerned with identifying object classes, their attributes, and operations
3. When implemented, objects are created from these classes and some control model used to coordinate object operations.

10.5.2 Invoice processing system



Advantages:

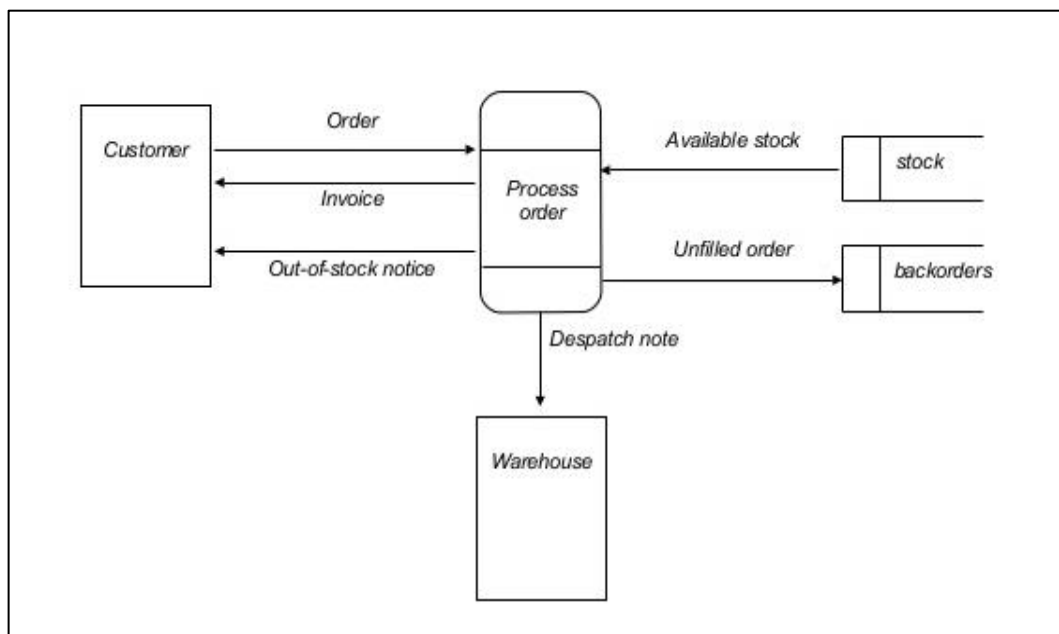
1. Objects are loosely coupled, the implementation of the objects are modify without affecting other objects.
2. Objects can be reused.
3. Direct implementation of architectural components.

Disadvantages:

1. To use services, objects must explicitly reference the name and the interface of other objects.
2. Interface change is required to satisfy proposed system changes, the effect of that change on all users of the changed object must be evaluated.

10.5.3 Function Oriented Pipelining or Data flow model:

1. Functional transformation processes their inputs to produce outputs.
2. Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
3. It may be referred to as a pipe and filter model.

**Fig: Data Flow Model**

10.6 Reference architectures

- Reference models are derived from a study of the application domain rather than from existing systems.
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated.
- OSI model is a layered model for communication systems

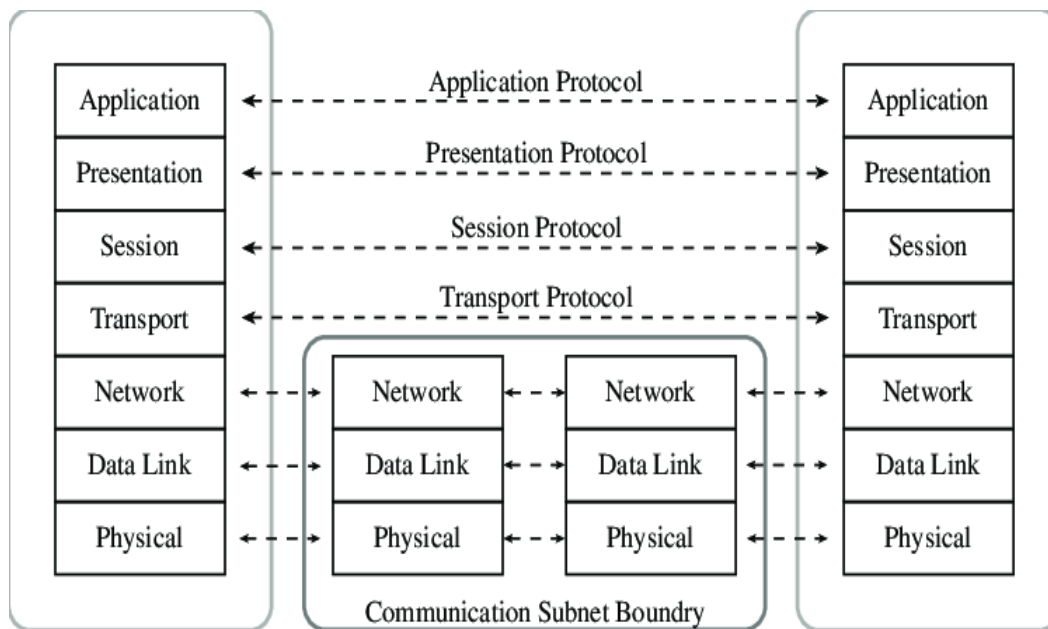


Fig: OSI Reference Model

- The software architect is responsible for deriving a structural system model, a control model, and a sub-system decomposition model.
- Large systems rarely conform to a single architectural model

10.7 Application architecture

The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re implementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

10.7.1 As a software designer, you can use models of application architectures in several Ways which are as follows.

1. As a starting point for the architectural design process If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. Of course, this will have to be specialized for the specific system being developed, but it is a good starting point for design.
2. As a design checklist If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.

3. As a way of organizing the work of the development team. The application architectures identify stable structural features of the system architectures and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.
4. As a means of assessing components for reuse If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.
5. As a vocabulary for talking about types of applications If you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications.

10.7.2 There are many types of application system, there are architecture of two type of application.

1. Transaction processing applications:

Transaction processing applications are database-cantered applications that process user requests for information and update the information in a database.

These are the most common type of inter active business systems. They are organized in such a way that user actions can't interfere with each other and the integrity of the database is maintained.

These systems include e-commerce system, information systems, and booking system

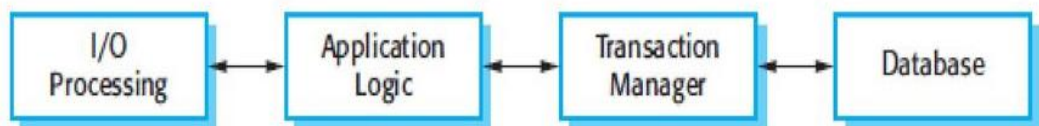


Fig: The structure of transaction processing application

2. Language processing systems:

Language processing systems are systems in which the user's intentions are expressed in a formal language (such as Java). The language processing system processes this language into an internal format and then interprets this internal representation.

The best-known language processing systems are compilers, which translate high-level language programs into machine code.

However, language processing systems are also used to interpret command languages for databases and information systems, and mark-up languages such as XML.

Transaction Processing Systems

Transaction processing system are designed to process user request for information from a database, or request to update database.

a database transaction is a sequence of operation that is treated as a single unit, all of the operation in a transaction have to be completed before the database changes and made permanent. This ensures that failure of operation within the transaction does not lead to inconsistency in the database.

Hey for a user perspective a transaction is any coherent sequence of operation that satisfy a goal, If the user transaction does not required the database to be changed then it may not be necessary to package this as a technical database transaction

For example, a transaction is a customer request to withdraw money from a bank account using an ATM.

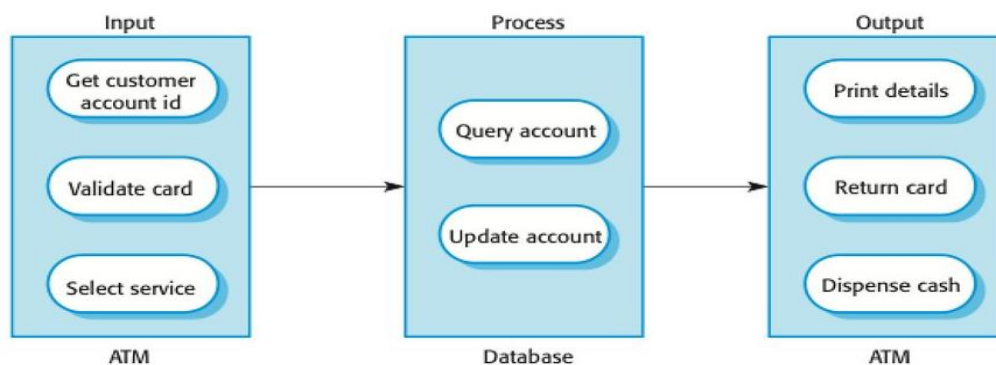


Fig: The Software architecture of ATM system

Transaction processing system may be organised as pipe and filter architecture with system components responsible for input, processing, and output.

The system is composed of 2 component cooperating software component the ATM software and the account processing software in the bank database server.

Summary

Requirement for a software system set out what the system should do and define constraints on its operation and implementation. The Specifications determining part for project success or failure.

Graded Question

1. What are the advantages of Architectural Design? Which factors are dependable during the design?
2. What are the three system organization styles of architectural design? Explain in brief.
3. What is the design perspective of architectural design? Explain.
4. What are the three system organization styles of architectural design? Explain in brief.
5. Explain Object oriented decomposition.

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach, Roger Pressman, Tata Mcgraw-hill , Seventh

USER INTERFACE DESIGN – RAPID SOFTWARE DEVELOPMENT

Unit Structure

- 11.0 Objectives
- 11.1 Introduction
- 11.2 Need of UI design
- 11.3 Design issues
 - 11.3.1 User interaction
 - 11.3.2 Information presentation
- 11.4 The UI design Process
- 11.5 User Analysis
 - 11.5.1 Analysis Techniques
- 11.6 User Interface Prototyping
- 11.7 Interface Evaluation
- 11.8 Agile Methods
- 11.9 Extreme Programming
 - 11.9.1 Testing in XP
 - 11.9.2 Pair programming
- 11.10 Rapid Application Development
- 11.11 Software Prototyping
- 11.12 Summary
- 11.13 Self-Assessment Questions
- 11.14 References

11.0 Objectives

The objective of this chapter is to introduce some aspects of user interface design that are important for software engineers. When you have read this chapter, you will:

- understand several user interface design principles.
- be introduced to several interaction styles and understand when these are most appropriate.
- understand when to use graphical and textual presentation of information.
- come to know what is involved in the principal activities in the use interface design process.
- Understand usability attributes and have been introduced to different approaches to interface evaluation.

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan driven development.
- know the key practices in extreme programming and how these relate to the general principles of agile methods.
- understand the importance of RAD
- know the concept of Software prototyping

11.1 Introduction

Computer system design encompasses a spectrum of activities from hardware design to user interface design. While specialists are often employed for hardware design and for the graphic design of web pages, only large organizations normally employ specialist interface designers for their application software. Therefore, software engineers must often take responsibility for user interface design as well as for the design of the software to implement that interface. Even when software designers and programmers are competent users of interface implementation technologies, such as Java's Swing classes (Elliott et al., 2(02) or XHTML (Musdano and Kennedy, 2002), the user interfaces they develop are often unattractive and inappropriate for their target users.

The focus here is, therefore, on the design products for user interfaces rather than the software that implements these facilities. Because of space limitations, it is considered only graphical user interfaces. It is not discussed about interfaces that require special (perhaps very simple) displays such as cell phones, DVD players, televisions, copiers and fax machines.

Careful user interface design is an essential part of the overall software design process. If a software system is to achieve its full potential, it is essential that its user interface should be designed to match the skills, experience, and expectations of its anticipated users. Good user interface design is critical for system dependability. Many so-called user errors are caused by the fact that user interfaces do not consider the capabilities of real users and their working environment. A poorly designed user interface means that users will probably be unable to access some of the system features, will make mistakes and will feel that the system hinders rather than helps them in achieving whatever they are using the system for.

11.2 Need of UI Design

When making user interface design decisions, you should consider the physical and mental capabilities of the people who use software. Human issues in detail discussed here but important factors that you should consider are:

1. People have a limited short-term memory-we can instantaneously remember about seven items of information (Miller, 1957). Therefore, if you present users with too much information at the same time, they may not be able to take it all in.

2. We all make mistakes, especially when we must handle too much information or are under stress. When systems go wrong and issue warning messages and alarms, this often puts more stress on users, thus increasing the chances that they will make operational errors.
3. We have a diverse range of physical capabilities. Some people see and hear better than others, some people are color-blind, and some are better than others at physical manipulation. You should not design for your own capabilities and assume that all other users will be able to cope.
4. We have different interaction preferences. Some people like to work with pictures, others with text. Direct manipulation is natural for some people, but others prefer a style of interaction that is based on issuing commands to the system.

Principle	Description
User familiarity	The interface should use terms and concepts drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to recover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system users.

Figure 11.1

These human factors are the basis for the design principles shown in Figure 11.1. These general principles are applicable to all user interface designs and should normally be instantiated as more detailed design guidelines for specific organizations or types of system. User interface design principles are covered in more detail by Dix, et al. (Dix, et al., 2004). Shneiderman (Shneiderman, 1998) gives a longer list of more specific user interface design guidelines.

The principle of user familiarity suggests that users should not be forced to adapt to an interface because it is convenient to implement. The interface should use terms that are familiar to the user, and the objects manipulated by the system should be directly related to the user's working environment. For example, if a system is designed for use by air traffic controllers, the objects manipulated should be aircraft, flight paths, beacons, and so on. Associated operations might be to increase

or reduce aircraft speed, adjust heading, and change height. The underlying implementation of the interface in terms of files and data structures should be hidden from the end user.

The principle of user interface consistency means that system commands and menus should have the same format, parameters should be passed to all commands in the same way, and command punctuation should be similar. Consistent interfaces reduce user learning time. Knowledge learned in one command or application is therefore applicable in other parts of the system or in related applications. Interface consistency across applications is also important. As far as possible, commands with similar meanings in different applications should be expressed in the same way.

Errors are often caused when the same keyboard command, such as 'Control-b' means different things in different systems. For example, in the word processor that is normally used, 'Control-b' means embolden text, but in the graphics program that is used to draw diagrams, 'Control-b' means move the selected object behind another object. The mistakes are made when using them together and sometimes try to embolden text in a diagram using the key combination. Then to get confused when the text disappears behind the enclosing object. You can normally avoid this kind of error if you follow the command key shortcuts defined by the operating system that you use.

This level of consistency is low-level. Interface designers should always try to achieve this in a user interface. Consistency at a higher level is also sometimes desirable. For Example, it may be appropriate to support the same operations (print, copy, etc.) on all types of system entities. However, Grodin (Grodin, 1989) points out that complete consistency is neither possible nor desirable. It may be sensible to implement deletion from a desktop by dragging entities into a trash can. It would be awkward to delete text in a word processor in this way. Unfortunately, the principles of user familiarity and user consistency are sometimes conflicting. Ideally, applications with common features should always use the same commands to access these features. However, this can conflict with user practice when systems are designed to support a particular type of user, such as graphic designers. These users may have evolved their own styles of interactions, terminology and operating conventions. These may clash with the interaction 'standards' that are appropriate to more general applications such as word processors.

The principle of minimal surprise is appropriate because people get very irritated when a system behaves in an unexpected way. As a system is used, users build a mental model of how the system works. If an action in one context causes a particular type of change, it is reasonable to expect that the same action in a different context it will cause a comparable change. If something completely different happens, the user is both surprised and confused.

Interface designers should therefore try to ensure that comparable actions have comparable effects. Surprises in user interfaces are often the result of the fact that many interfaces are modeled. This means that there are several modes of working (e.g., viewing mode and editing mode), and the effect of a command is different depending on the mode. It is very important that, when designing an interface, you include a visual indicator showing the user the current mode.

The principle of recoverability is important because users inevitably make mistakes when using a system. The interface design can minimize these mistakes (e.g., using menus means avoids typing mistakes), but mistakes can never be completely eliminated. Consequently, you should include interface facilities that allow users to recover from their mistakes. These can be of three kinds:

1. Confirmation of destructive actions
If a user specifies an action that is potentially destructive, the system should ask the user to confirm that this is really what is wanted before destroying any information.
2. The provision of an undo facility
Undo restores the system to a state before the action occurred. Multiple levels of undo are useful because users don't always recognize immediately that a mistake has been made.
3. Checkpointing
Checkpointing involves saving the state of a system at periodic intervals and allowing the system to restart from the last checkpoint. Then, when mistakes occur, users can go back to a previous state and start again. Many systems now include checkpointing to cope with system failures but, paradoxically, they don't allow system users to use them to recover from their own mistakes.

A related principle is the principle of user assistance. Interfaces should have built in user assistance or help facilities. These should be integrated with the system and should provide different levels of help and advice. Levels should range from basic information on getting started to a full description of system facilities. Help systems should be structured so that users are not overwhelmed with information when they ask for help.

The principle of user diversity recognizes that, for many interactive systems, there may be different types of users. Some will be casual users who interact occasionally with the system while others may be power users who use the system for several hours each day.

Casual users need interfaces that provide guidance, but power users require shortcuts so that they can interact as quickly as possible. Furthermore, users may

suffer from disabilities of various types and, if possible, the interface should be adaptable to cope with these. Therefore, you might include facilities to display enlarged text, to replace sound with text, to produce very large buttons and so on. This reflects the notion of Universal Design (UD) (Preiser and Ostoff, 2001), a design philosophy whose goal is to avoid excluding users because of thoughtless design choices.

The principle of recognizing user diversity can conflict with the other interface design principles, since some users may prefer very rapid interaction over, for example, user interface consistency. Similarly, the level of user guidance required can be radically different for different users, and it may be impossible to develop support that is suitable for all types of users. You therefore have to make compromises to reconcile the needs of these users.

11.3 Design issues

A coherent user interface must integrate user interaction and information presentation. This can be difficult because the designer must find a compromise between the most appropriate styles of interaction and presentation for the application, the background and experience of the system users, and the equipment that is available.

11.3.1 User interaction

User interaction means issuing commands and associated data to the computer system. On early computers, the only way to do this was through a command-line interface, and a special-purpose language was used to communicate with the machine. However, this was geared to expert users and a number of approaches have now evolved that are easier to use. Shneiderman (Shneiderman, 1998) has classified these forms of interaction into five primary styles:

- Direct manipulation
- The user interacts directly with objects on the screen. Direct manipulation usually involves a pointing device (a mouse, a stylus, a trackball or, on touch screens, a finger) that indicates the object to be manipulated and the action, which specifies what should be done with that object. For example, to delete a file, you may click on an icon representing that file and drag it to a trash can icon.
- Menu selection
- The user selects a command from a list of possibilities (a menu). The user may also select another screen object by direct manipulation, and the

command operates on that object. In this approach, to delete a file, you would select the file icon then select the delete command.

- **Form fill-in**
- The user fills in the fields of a form. Some fields may have associated menus, and the form may have action 'buttons' that, when pressed, cause some action to be initiated. You would not normally use this approach to implement the interface to operations such as file deletion. Doing so would involve filling in the name of the file on the form then 'pressing' a delete button.
- **Command language**
- The user issues a special command and associated parameters to instruct the system what to do. To delete a file, you would type a delete command with the filename as a parameter.
- **Natural language**
- The user issues a command in natural language. This is usually a front end to a command language; the natural language is parsed and translated to system commands. To delete a file, you might type 'delete the file named xxx'.

Each of these styles of interaction has advantages and disadvantages and is best suited to a particular type of application and user (Shneiderman, 1998). Figure 11.2 shows the main advantages and disadvantages of these styles and suggests types of applications where they might be used. Of course, these interaction styles may be mixed, with several styles used in the same application.

Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement Only suitable where there is a visual metaphor for tasks and objects	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users Can become complex if many menu options	Most general-purpose systems
Form fill-in Easy to learn Checkable	Simple data entry	Takes up a lot of screen space Causes problems where user options do not match the form fields	Stock control Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems Command and control systems
Natural language	Accessible to casual users Easily extended	Requires more typing Natural language understanding systems are unreliable	Information retrieval systems

Figure 11.2 Advantages and disadvantages of interaction styles

For example, Microsoft Windows supports direct manipulation of the iconic representation of files and directories, menu-based command selection, and for commands such as configuration commands, the user must fill in a special-purpose form that is presented to them. In principle, it should be possible to separate the interaction style from the underlying entities that are manipulated through the user interface. This was the basis of the Seeheim model (Pfaff and ten Hagen, 1985) of user interface management. In this model, the presentation of information, the dialogue management and the application are separate.

In reality, this model is more of an ideal than practical, but it is certainly possible to have separate interfaces for different classes of users (casual users and experienced users, say) that interact with the same underlying system.

This is illustrated in Figure 11.3, which shows a command language interface and a graphical interface to an underlying operating system such as Linux. Web-based user interfaces are based on the support provided by HTML or XHTML (the page description languages used for web pages) along with languages such as Java, which can associate programs with components on a page. Because these web-based interlaces are usually designed for casual users, they mostly use forms-based interfaces.

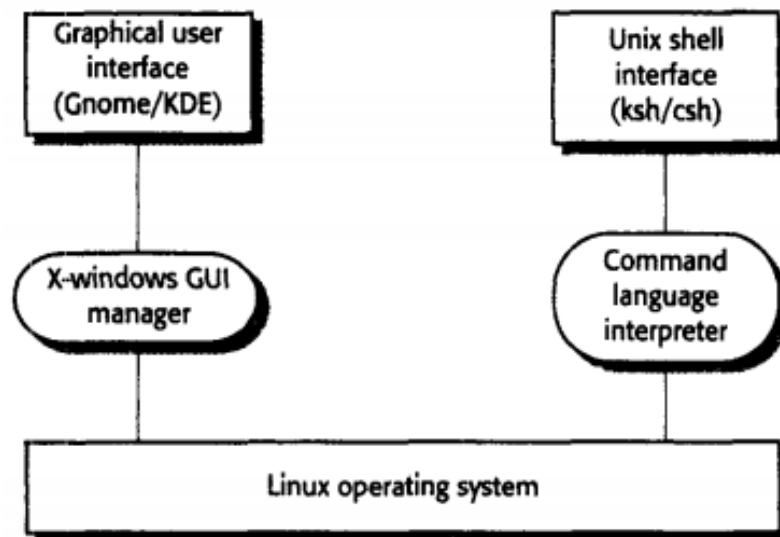
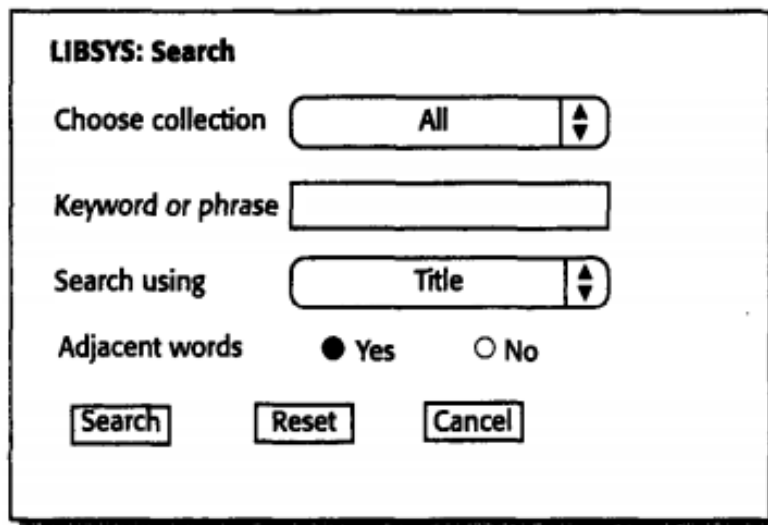


Figure 11.3

It is possible to construct direct manipulation interlaces on the web, but this is a complex programming task. Furthermore, because of the range of experience of web users and the fact that they come from many different cultures, it is difficult to establish a user interface metaphor for direct interaction that is universally acceptable.

To illustrate the design of web-based user interaction, it is discussed the approach used in the LIBSYS system where users can access documents from other libraries. There are two, fundamental operations that need to be supported:

1. Document search where users use the search facilities to find the documents that they need.
 2. Document request where users request that the document be delivered to their local machine or server for printing
- The LIBSYS user interface is implemented using a web browser, so, given that users must supply information to the system such as the document identifier, their name and their authorization details, it makes sense to use a forms-based interlace. Figure 11.4 shows a possible interlace design for the search component of the system. In form-based interfaces, the user supplies all of the information required then initiates, the action by pressing a button. Forms fields can be menus, free-text input fields or radio buttons. In the LIBSYS example, a user chooses the collection to search from a menu of collections that can be accessed ('All' is the default, meaning search all collections) and types the search phrase into a free-text input field. The user chooses the field of the library record from a menu ('Title' is the default) and selects a radio button to indicate whether the search terms should be adjacent in the record.



LIBSYS: Search

Choose collection

Keyword or phrase

Search using

Adjacent words ☒ Yes ☐ No

Figure 11.4 A forms-based interface to the LIBSYS system

11.3.2 Information presentation

All interactive systems have to provide some way of presenting information to users. The information presentation may simply be a direct representation of the input information (e.g., text in a word processor) or it may present the information graphically. A good design guideline is to keep the software required for

information presentation separate from the information itself. Separating the presentation system from the data allows us to change the representation on the user's screen without having to change the underlying computational system. This is illustrated in Figure 11.5. The MVC approach (Figure 11.6), first made widely available in Smalltalk (Goldberg and Robson, 1983), is an effective way to support multiple presentations of data. Users can interact with each presentation in a style that is appropriate to the presentation. The data to be displayed is encapsulated in a model object. Each model object may have a number of separate view objects associated with it where each view is a different display representation of the model. Each view has an associated controller object that handles user input and device interaction. Therefore, a model that represents numeric data may have a view that represents the data as a histogram and a view that presents the data as a table. The model may be edited by changing the values in the table or by lengthening or shortening the bars in the histogram.

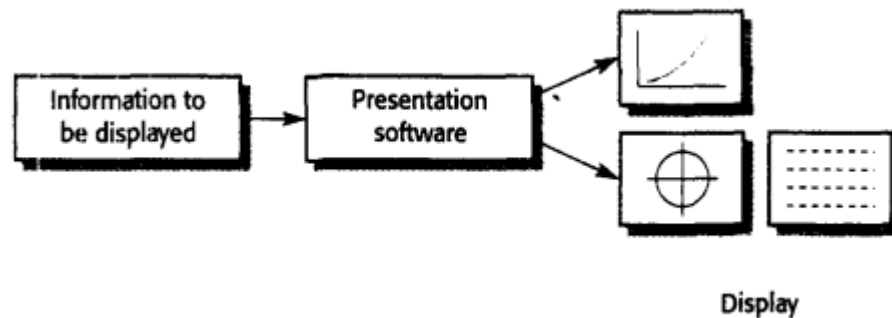


Figure 11.5

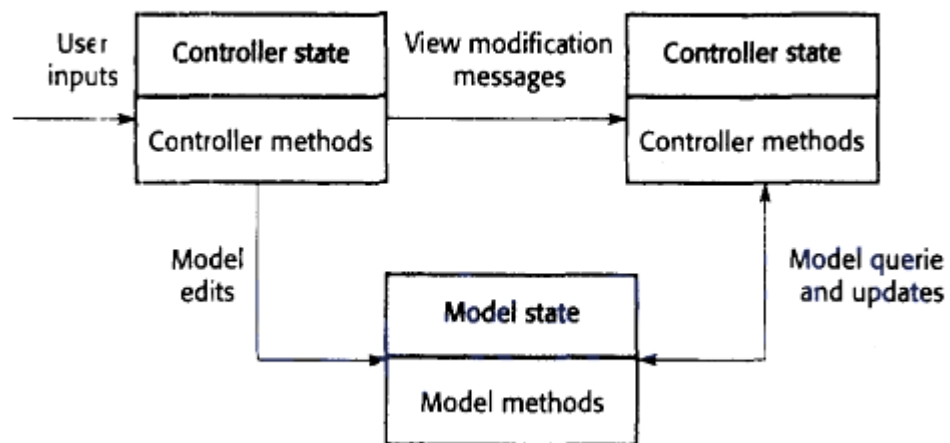


Figure 11.6

You should not assume that using graphics makes your display more interesting. Graphics take up valuable screen space (a major issue with portable devices) and can take a long time to download if the user is working over a slow, dial-up connection. Information that does not change during a session may be presented either graphically or as text depending on the application. Textual presentation takes up less screen space but cannot be read at a glance. You should distinguish information that does not change from dynamic information by using a different presentation style. For example, you could present all static information in a particular font or color, or you could associate a 'static information' icon with it. You should use text to present information when precise information is required and the information changes relatively slowly. If the data changes quickly or if the relationships between data rather than the precise data values are significant, then you should present the information graphically.

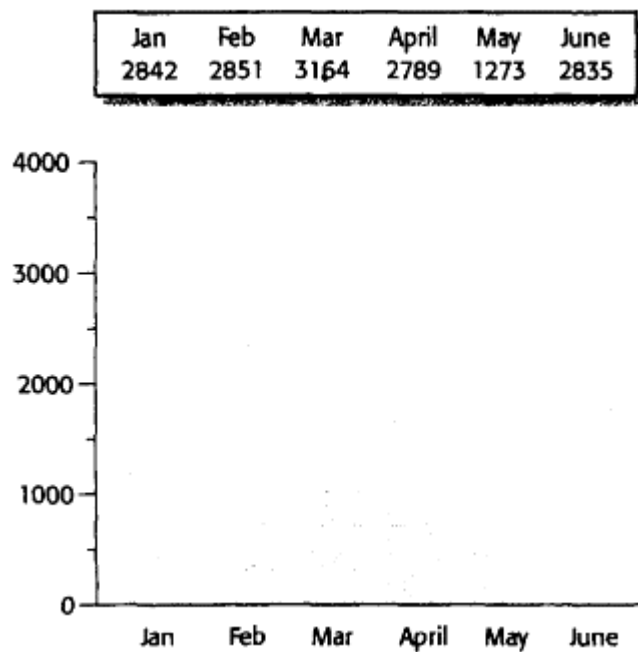


Figure 11.7 Alternative information presentations

For example, consider a system that records and summarizes the sales figures for a company on a monthly basis. Figure 11.7 illustrates how the same information can be presented as text or in a graphical form. Managers studying sales figures are usually more interested in trends or anomalous figures rather than precise values. Graphical presentation of this information, as a histogram, makes the anomalous figures in March and May stand out from the others. Figure 11.7 also illustrates how textual presentation takes less space than a graphical representation of the same information.

In control rooms or instrument panels such as those on a car dashboard, the information that is to be presented represents the state of some other system (e.g., the altitude of an aircraft) and is changing all the time. A constantly changing digital display can be confusing and irritating as readers can't read and assimilate the information before it changes. Such dynamically varying numeric information is therefore best presented graphically using an analogue representation. The graphical display can be supplemented if necessary with a precise digital display. Different ways of presenting dynamic numeric information are shown in Figure 11.8. Continuous analogue displays give the viewer some sense of relative value. In Figure 11.9, the values of temperature and pressure are approximately the same. However, the graphical display shows that temperature is close to its maximum value whereas pressure has not reached 25% of its maximum. With only a digital value, the viewer must know the maximum values and mentally compute the relative state of the reading. The extra thinking time required can lead to human errors in stressful situations when problems occur and operator displays may be showing abnormal readings.

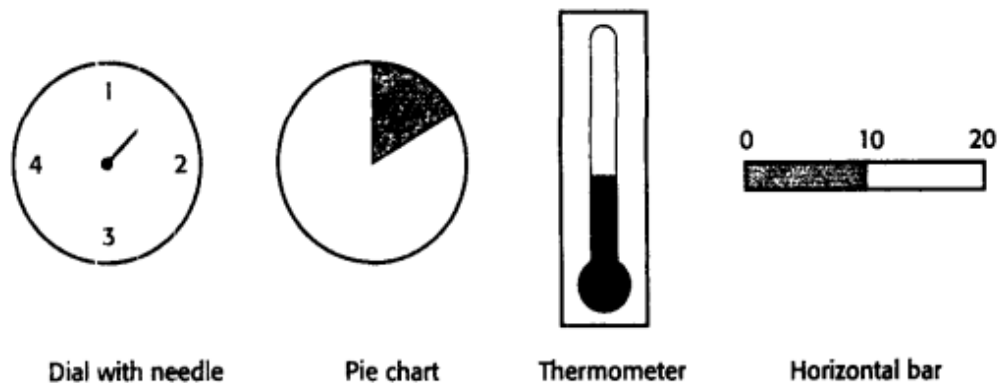


Figure 11.8 Methods of presenting dynamically varying numeric information

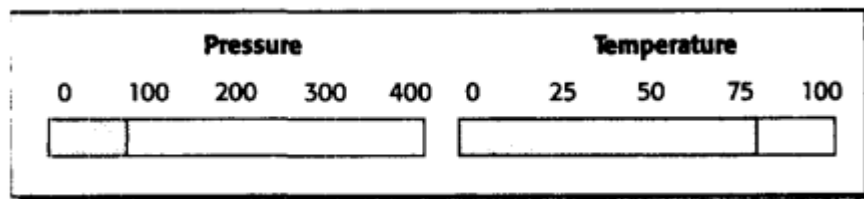


Figure 11.9 Graphical information display showing relative values

When large amounts of information have to be presented, abstract visualizations that link related data items may be used. This can expose relationships that are not obvious from the raw data. You should be aware of the possibilities of visualization, especially when the system user interface must represent physical entities.

Examples of data visualizations are:

1. Weather information, gathered from several sources, is shown as a weather map with isobars, weather fronts, and so on.
2. The state of a telephone network is displayed graphically as a linked set of nodes in a network management center.
3. The state of a chemical plant is visualized by showing pressures and temperatures in a linked set of tanks and pipes.
4. A model of a molecule is displayed and manipulated in three dimensions using a virtual reality system.
5. A set of web pages is displayed as a hyperbolic tree (Lamping et al., 1995).

Shneiderman (Shneiderman, 1998) offers a good overview of approaches to visualization as well as identifies classes of visualization that may be used. These include visualizing data. Using two- and three-dimensional presentations and as trees or networks. Most of these are concerned with the display of large amounts of information managed on a computer. However, the most common use of visualization in user interfaces is to represent some physical structure such as the molecular structure of a new drug, the links in a telecommunications network and so on. Three- dimensional presentations that may use special virtual reality equipment are particularly effective in product visualizations. Direct manipulation of these visualizations is a very effective way to interact with the data. In addition to the style of information presentation, you should think carefully about how color is used in the interface. Color can improve user interfaces by helping users understand and manage complexity. However, it is easy to misuse color and to create user interfaces that are visually unattractive and error prone. Shneiderman gives 14 key guidelines for the effective use of color in user interfaces. The most important of these are:

1. Limit the number of colors employed and be conservative how these are used
You should not use more than four or five separate colors in a window and no more than seven in a system interface. If you use too many, or if they are too bright, the display may be confusing. Some users may find masses of color disturbing and visually tiring. User confusion is also possible if colors are used inconsistently.
2. Use color change to show a change in system status
If a display changes color, this should mean that a significant event has occurred. Thus, in a fuel gauge, you could use a change of color to indicate that fuel is running low. Color highlighting is particularly important in complex displays where hundreds of distinct entities may be displayed.
3. Use color coding to support the task users are trying to perform
If they have to identify anomalous instances, highlight these instances; if similarities are also to be discovered, highlight these using a different color.

4. Use color coding in a thoughtful and consistent way
If one part of a system displays error messages in red (say), all other parts should do likewise. Red should not be used for anything else. If it is, the user may interpret the red display as an error message.
5. Be careful about color pairings
Because of the physiology of the eye, people cannot focus on red and blue simultaneously. Eyestrain is a likely consequence of a red on blue display. Other color combinations may also be visually disturbing or difficult to read.

In general, you should use color for highlighting, but you should not associate meanings with colors. About 10% of men are color-blind and may misinterpret the meaning. Human color perceptions are different, and there are different conventions in different professions about the meaning of particular colors. Users with different backgrounds may unconsciously interpret the same color in different ways. For example, to a driver, red usually means danger. However, to a chemist, red means hot. As well as presenting application information, systems also communicate with users through messages that give information about errors and the system state. A user's first experience of a software system may be when the system presents an error message. Inexperienced users may start work, make an initial error and immediately have to understand the resulting error message. This can be difficult enough for skilled software engineers. It is often impossible for inexperienced or casual system users. Factors that you should take into account when designing system messages are shown in Figure 11.10.

Factor	Description
Context	Wherever possible, the messages generated by the system should reflect the current user context. As far as is possible, the system should be aware of what the user is doing and should generate messages that are relevant to their current activity
Experience	As users become familiar with a system they become irritated by long, 'meaningful' messages. However, beginners find it difficult to understand short, terse statements of a problem. You should provide both types of message and allow the user to control message conciseness.
Skill level	Messages should be tailored to the users' skills as well as their experience. Messages for the different classes of users may be expressed in different ways depending on the terminology that is familiar to the reader.
Style	Messages should be positive rather than negative. They should use the active rather than the passive mode of address. They should never be insulting or try to be funny.
Culture	Wherever possible, the designer of messages should be familiar with the culture of the country where the system is sold. There are distinct cultural differences between Europe, Asia and America. A suitable message for one culture might be unacceptable in another.

Figure 11.10 Design factors in message wording

You should anticipate the background and experience of users when designing

error messages. For example, say a system user is a nurse in an intensive-care ward in a hospital. Patient monitoring is carried out by a computer system. To view a patient's current state (heart rate, temperature, etc.), the nurse selects 'display' from a menu and inputs the patient's name in the box, as shown in Figure 11.11. In this case, let's assume that the nurse has misspelled the patient's name and has typed 'MacDonald' instead of 'McDonald'. The system generates an error message. Error messages should always be polite, concise, consistent and constructive. They must not be abusive and should not have associated beeps or other noises that might embarrass the user. Wherever possible, the message should suggest how the error might be corrected. The error message should be linked to a context-sensitive online help system. Figure 11.12 shows examples of good and bad error messages. The left-hand message is badly designed. It is negative (it accuses the user of making an error), it is not tailored to the user's skill and experience level, and it does not take context information into account.

Figure 11.11 An input text box used by a nurse

Figure 11.12 System and user-oriented error messages

It does not suggest how the situation might be rectified. It uses system-specific terms (patient id) rather than user-oriented language. The right-hand message is better. It is positive, implying that the problem is a system rather than a user problem. It identifies the problem in the nurse's terms and offers an easy way to correct the mistake by pressing a single button. The help system is available if required.

11.4 The UI design process

User interface (UI) design is an iterative process where users interact with designers and interface prototypes to decide on the features, organization and the look and feel of the system user interface. Sometimes, the interface is separately prototyped in parallel with other software engineering activities. More commonly, especially where iterative development is used, the user interface design proceeds incrementally as the software is developed. In both cases, however, before you start programming, you should have developed and, ideally, tested some paper-based designs. The overall UI design process is illustrated in Figure 11.13. There are three core activities in this process:

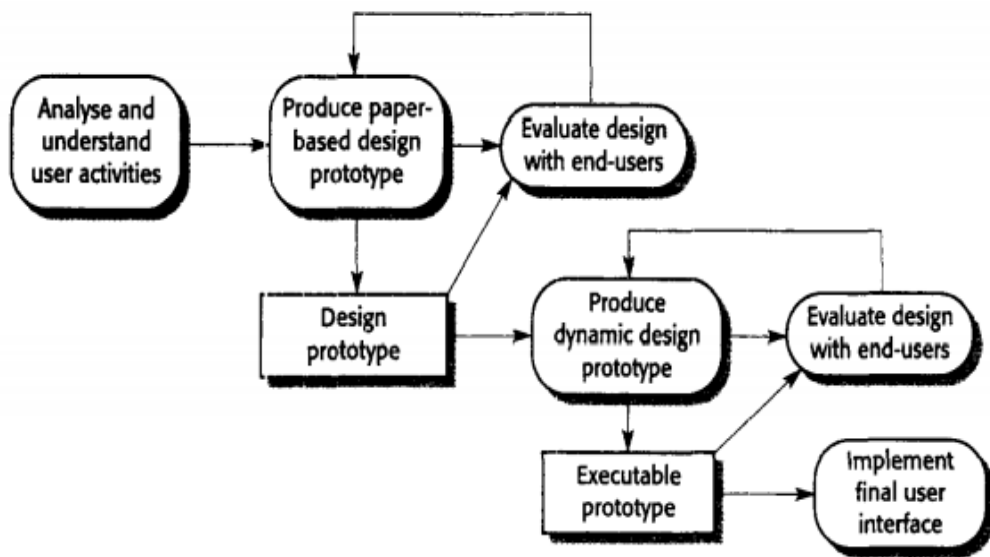


Figure 11.13 The UI design process

1. User analysis

In the user analysis process, you develop an understanding of the tasks that users do, their working environment, the other systems that they use, how they interact with other people in their work and so on. For products with a diverse range of users, you have to try to develop this understanding through focus groups, trials with potential users and similar exercises.

2. System-prototyping

User interface design and development is an iterative process. Although users may talk about the facilities they need from an interface, it is very difficult for them to be specific until they see something tangible. Therefore, you have to develop prototype systems and expose them to users, who call. then guide the evolution of the interface.

3. Interface evaluation

Although you will obviously have discussions with users during the prototyping process, you should also have a more formalized evaluation activity where you collect information about the users actual experience with the interface.

Radhika is a religious studies student writing an essay on Indian architecture and how it has been influenced by religious practices. To help her understand this, she would like to access pictures of details on notable buildings but cannot find anything in her local library. She approaches the subject librarian to discuss her needs and he suggests search terms that she might use. He also suggests libraries in Mumbai and London that might have this material, so he and Radhika log on to the library catalogues & search using these terms. They find some source material and place a request for photocopies of the pictures with architectural details, to be posted directly to Radhika

Figure 11.14 A library interaction scenario

11.5 User analysis

A critical UI design activity is the analyses of the user activities that are to be supported by the computer system. If you don't understand what users want to do with a system, then you have no realistic prospect of designing an effective user interface. To develop this understanding, you may use techniques such as task analysis, ethnographic studies, user interviews and observations or, commonly, a mixture of all of these.

A challenge for engineers involved in user analysis is to find a way to describe user analyses so that they communicate the essence of the tasks to other designers and to the users themselves. Notations such as UML sequence charts may be able to describe user interactions and are ideal for communicating with software engineers. However, other users may think of these charts as too technical and will not try to understand them. Because it is very important to engage users in the design process, you therefore usually have to develop natural language scenarios to describe user activities. Figure 11.14 is an example of a natural language scenario that might have been developed during the specification and design process for the LIBSYS system. It describes a situation where LIBSYS does not exist and where a student needs to retrieve information from another library. From this scenario, the designer can see a number of requirements:

1. Users might not be aware of appropriate search terms. They may need to access ways of helping them choose search terms.
2. Users have to be able to select collections to search.
3. Users need to be able to carry out searches and request copies of relevant material.

You should not expect user analysis to generate very specific user interface requirements. Normally, the analysis helps you understand the needs and concerns of the system users.

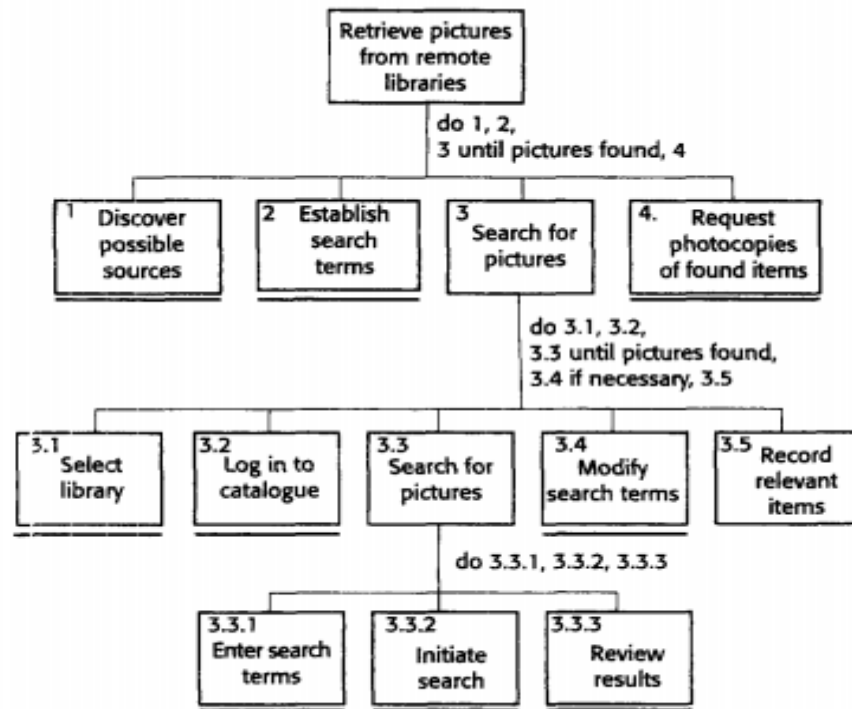


Figure 11.15

As you become more aware of how they work, their concerns and their constraints, your design can take these into account. This means that your initial designs (which you will refine through prototyping anyway) are more likely to be acceptable to users and so convince them to become engaged in the process of design refinement.

11.5.1 Analysis Techniques

There are various forms of task analysis (Diaper, 1989), but the most used is Hierarchical Task Analysis (HTA). HTA was originally developed to help with writing user manuals, but it can also be used to identify what users do to achieve some goal. In HTA, a high-level task is broken down into subtasks, and plans are identified that specify what might happen in a specific situation. Starting with a user goal, you draw a hierarchy showing what has to be done to achieve that goal. Figure 11.15 illustrates this approach using the library scenario introduced in Figure 11.14. In the HTA notation, a line under a box normally indicates that it will not be decomposed into more detailed subtasks.

The advantage of HTA over natural language scenarios is that it forces you to consider each of the tasks and to decide whether these should be decomposed. With

natural language scenarios, it is easy to miss important tasks. Scenarios also become long and boring to read if you want to add a lot of detail to them. The problem with this approach to describing user tasks is that it is best suited to tasks that are sequential processes. The notation becomes awkward when you try to model tasks that involve interleaved or concurrent activities or that involve a very large number of subtasks.

Furthermore, HTA does not record why tasks are done in a particular way or constraints on the user processes. You can get a partial view of user activities from HTA, but you need additional information to develop a fuller understanding of the UI design requirements. Normally, you collect information for HTA through observing and interviewing users. In this interviewing process, you can collect some of this additional information and record it alongside the task analyses. When interviewing to discover what users actually do, you should design interviews so that users can provide any information that they (rather than you) feel is relevant. This means you should not stick rigidly to prepared list of questions. Rather, your questions should be open ended and should encourage users to tell you why they do things as well as what they actually do. Interviewing, of course, is not just a way of gathering information for task analysis it is a general information-gathering technique.

You may decide to supplement individual interviews with group interviews or focus groups. The advantage of using focus groups is that users stimulate each other to provide information and may end up discussing different ways that they have developed of using systems. Task analysis focuses on how individuals work but, of course, most work is actually cooperative. People work together to achieve a goal, and users find it difficult to discuss how this cooperation actually takes place. Therefore, direct observation of how users work and use computer-based systems is an important additional technique of user analysis.

Air traffic control involves a number of control 'suites' where the suites controlling adjacent sectors of airspace are physically located next to each other. Flights in a sector are represented by paper strips that are fitted into wooden racks in an order that reflects their position in the sector. If there are not enough slots in the rack (i.e. when the airspace is very busy), controllers spread the strips out on the desk in front of the rack. When we were observing controllers, we noticed that controllers regularly glanced at the strip racks in the adjacent sector. We pointed this out to them and asked them why they did this. They replied that, when the adjacent controller has strips on his or her desk, then this means that a lot of flights will be entering their sector. They therefore tried to increase the speed of aircraft in the sector to 'clear space' for the incoming aircraft.

Figure 11.16 A report of observations of air traffic control

1. Controllers had to be able to see all flights in a sector (this was why they spread strip: out on the desk). Therefore, we should avoid using scrolling displays where flights disappeared off the top or bottom of the display.
2. The interface should have some way of telling controllers how many flights are in adjacent sectors so that controllers can plan their workload. Checking adjacent sectors was an automatic controller action and it is very likely that they would not have mentioned this in discussions of the ATC process. It was only through direct observation that we discovered these important requirements.

None of these user analysis techniques, on their own, give you a complete picture of what users actually do. They are complementary approaches that you should use together to help you understand what users do and get insights into what might be an appropriate user interface design.

11.6 User interface prototyping

Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good! enough for expressing user interface requirements. Evolutionary or exploratory prototyping with end-user involvement is the only practical way to design and develop graphical user interfaces for software systems. Involving the user in the design and development process is an essential aspect of user-centred design (Norman and Draper, 1986), a design philosophy for interactive systems. The aim of prototyping is to allow users to gain direct experience with the interface. Most of us find it difficult to think abstractly about a user interface and to explain exactly what we want. However, when we are presented with examples, it is easy to identify the characteristics that we like and dislike.

Ideally, when you are prototyping a user interface, you should adopt a two-stage prototyping process:

1. Very early in the process, you should develop paper prototypes-mock-ups of screen designs-and walk through these with end-users.
2. You then refine your design and develop increasingly sophisticated automated prototypes, then make them available to users for testing and activity simulation.

Paper prototyping is a cheap and surprisingly effective approach to prototype development (Snyder, 2003). You don't need to develop any executable software and the designs don't have to be drawn to professional standards. You can draw paper versions of the system screens that user interact with and design a set of

scenarios describing how the system might be used. As a scenario progresses, you sketch the information that would be displayed and the options available to users. You then work through these scenarios with users to simulate how the system might be used. This is an effective way to get users' initial reactions to an interface design, the information they need from the system and how they would normally interact with the system.

Alternatively, you can use a storyboarding technique to present the interface design. A storyboard is a series of sketches that illustrate a sequence of interactions. This is less hands-on but can be more convenient when presenting the interface proposals to groups rather than individuals. After initial experiments with a paper prototype, you should implement a software prototype of the interface design.

The problem, of course, is that you need to have some system functionality with which the users can interact. If you are prototyping the UI very early in the system development process, this may not be available. To get around this problem, you can use 'Wizard of Oz' prototyping (see the web page for an explanation if you haven't seen the film). In this approach, users interact with what appears to be a computer system, but their inputs are actually channeled to a hidden person who simulates the system's responses. They can do this directly or by using some other system to compute the required responses.

In this case, you don't need to have any executable software apart from the proposed user interface. There are three approaches that you can use for user interface prototyping:

1. Script-driven approach

If you simply need to explore ideas with users, you can use a script-driven approach such as you'd find in Macromedia Director. In this approach, you create screens with visual elements, such as buttons and menus, and associate a script with these elements. When the user interacts with these screens, the script is executed and the next screen is presented, showing them the results of their actions. There is no application logic involved.

2. Visual programming languages

3. Visual programming languages, such as Visual Basic, incorporate a powerful development environment, access to a range of reusable objects and a user-interface development system that allows interfaces to be created quickly, with components and scripts associated with interface objects. These solutions, based on web browsers and languages such as Java, offer a ready-made user interface. You add functionality by associating segments of Java

programs with the information to be displayed. These segments (called applets) are executed automatically when the page is loaded into the browser. This approach is a fast way to develop user interface prototypes, but there are inherent restrictions imposed by the browser and the Java security model.

Prototyping is obviously closely associated with interface evaluation. Formal evaluation is unlikely to be cost-effective for early prototypes, so what you are trying to achieve at this stage is a 'formative evaluation' where you look for ways in which the interface can be improved.

11.7 Interface evaluation

Interface evaluation is the process of assessing the usability of an interface and checking that it meets user requirements. Therefore, it should be part of the normal verification and validation process for software systems. For example, in a learnability specification, you might state that an operator who is familiar with the work supported should be able to use 80% of the system functionality after a three-hour training session. However, it is more common to specify usability (if it is specified at all) qualitatively rather than using metrics. You therefore usually have to use your judgement and experience in interface evaluation.

Systematic evaluation of a user interface design can be an expensive process involving cognitive scientists and graphics designers. You may have to design and carry out a statistically significant number of experiments with typical users. You may need to use specially constructed laboratories fitted with monitoring equipment. A user interface evaluation of this kind is economically unrealistic for systems developed by small organizations with limited resources.

Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

Figure 11.17 Usability attributes

There are several simpler, less expensive techniques of user interface evaluation that can identify particular user interface design deficiencies:

1. Questionnaires that collect information about what users thought of the interface.
2. Observation of users at work with the system and 'thinking aloud' about how they are trying to use the system to accomplish some tasks.
3. Video 'snapshots' of typical system use.
4. The inclusion in the software of code which collects information about the most used facilities and the most common errors.

Surveying users by questionnaire is a relatively cheap way to evaluate an interface. The questions should be precise rather than general. It is no use asking questions such as 'Please comment on the usability of the interface' as the responses will probably vary so much that you won't see any common trend. Rather, specific questions such as 'Please rate the understandability of the error messages on a scale from 1 to 5. A rating of 1 means very clear and 5 means incomprehensible' are better. They are both easier to answer and more likely to provide useful information to improve the interface. Users should be asked to rate their own experience and background when filling in the questionnaire. This allows the designer to find out whether users from any background have problems with the interface.

Questionnaires can even be used before any executable system is available if a paper mock-up of the interface is constructed and evaluated. Observation-based evaluation simply involves watching users as they use a system, looking at the facilities used, the errors made and so on. This can be supplemented by 'think aloud' sessions where users talk about what they are trying to achieve, how they understand the system and how they are trying to use the system to accomplish their objectives.

Relatively low-cost video equipment means that you can record user sessions for later analysis. Complete video analysis is expensive and requires a specially equipped evaluation suite with several cameras focused on the user and on the screen. However, video recording of selected user operations can be helpful in detecting problems. Other evaluation methods must be used to find out which operations cause user difficulties.

Analysis of recordings allows the designer to find out whether the interface requires too much hand movement (a problem with some systems is that users must regularly move their hand from keyboard to mouse) and to see whether unnatural eye movements are necessary. An interface that requires many shifts of focus may

mean that the user makes more errors and misses parts of the display. Instrumenting code to collect usage statistics allows interfaces to be improved in a number of ways.

The most common operations can be detected. The interface can be reorganized so that these are the fastest to select. For example, if pop-up or pull-down menus are used, the most frequent operations should be at the top of the menu and destructive operations towards the bottom. Code instrumentation also allows error-prone commands to be detected and modified. Finally, it is easy to give users a 'gripe' command that they can use to pass messages to the tool designer. This makes users feel that their views are being considered.

The interface designer and other engineers can gain rapid feedback about individual problems. None of these relatively simple approaches to user interface evaluation is foolproof and they are unlikely to detect all user interface problems. However, the techniques can be used with a group of volunteers before a system is released without a large outlay of resources. Many of the worst problems of the user interface design can then be discovered and corrected.

Rapid Software Development

Businesses now operate in a global, rapidly changing environment. They must respond to new opportunities and markets, changing economic conditions, and the emergence of competing products and services. Software is part of almost all business operations so new software is developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid development and delivery are therefore now often the most critical requirement for software systems. In fact, many businesses are willing to trade off software quality and compromise on requirements to achieve faster deployment of the software that they need.

Because these businesses are operating in a changing environment, it is often practically impossible to derive a complete set of stable software requirements. The initial requirements inevitably change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, the requirements are likely to change quickly and unpredictably due to external factors. The software may then be out of date when it is delivered. Software development processes that plan on completely specifying the requirements and then designing, building, and testing the system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to

be reworked and retested. Therefore, a conventional waterfall or specification-based process is usually prolonged, and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, a plan-driven approach is the right one. However, in a fast-moving business environment, this can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential. The need for rapid system development and processes that can handle changing requirements has been recognized for some time. IBM introduced incremental development in the 1980s (Mills et al., 1980). The introduction of so-called fourth generation languages, also in the 1980s, supported the idea of quickly developing and delivering software (Martin, 1981).

However, the notion really took off in the late 1990s with the development of the notion of agile approaches such as DSDM (Stapleton, 1997), Scrum (Schwaber and Beedle, 2001), and extreme programming (Beck, 1999; Beck, 2000). Rapid software development processes are designed to produce useful software quickly. The software is not developed as a single unit but as a series of increments, with each increment including new system functionality. Although there are many approaches to rapid software development, they share some fundamental characteristics:

1. The processes of specification, design, and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document only defines the most important characteristics of the system.
2. The system is developed in a series of versions. End-users and other system stakeholders are involved in specifying and evaluating each version. They may propose changes to the software and new requirements that should be implemented in a later version of the system.
3. System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface. The system may then generate a web-based interface for a browser or an interface for a specific platform such as Microsoft Windows.

11.8 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, the use of analysis and design methods supported by CASE tools and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long lived software systems such as aerospace and government systems. This software was developed by large teams working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. These plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design have to change with the program. Dissatisfaction with these heavyweight approaches to software engineering led a number of software developers in the 1990s to propose new ‘agile methods.

These allowed the development team to focus on the software itself rather than on its design and documentation. Agile methods universally rely on an incremental approach to software specification, development, and delivery. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut down on process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used. The philosophy behind agile methods is reflected in the agile manifesto that was agreed on by many of the leading developers of these methods. This manifesto states:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value: Individuals and interactions

over processes and tools Working software over comprehensive documentation Customer collaboration over contract negotiation Responding to change over following a plan That is, while there is value in the items on the right, we value the items on the left more.

Although these agile methods are all based around the notion of incremental development and delivery, they propose different processes to achieve this. However, they share a set of principles, based on the agile manifesto, and so have much in common. These principles are shown in Figure 11.18.

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Figure 11.18 The principles of agile methods

Agile methods have been very successful for some types of system development:

- Product development where a software company is developing a small or medium-sized product for sale.
- Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.

In practice, the principles underlying agile methods are sometimes difficult to realize:

1. Although the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Frequently, the customer representatives are subject to other pressures and cannot take full part in the software development.

2. Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore not interact well with other team members.
3. Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.
5. Many organizations, especially large companies, have spent years changing their culture so that processes are defined and followed. It is difficult for them to move to a working model in which processes are informal and defined by development teams.

Another non-technical problem—that is a general problem with incremental development and delivery—occurs when the system customer uses an outside organization for system development. The software requirements document is usually part of the contract between the customer and the supplier. Because incremental specification is inherent in agile methods, writing contracts for this type of development may be difficult. Consequently, agile methods must rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements. So long as all goes well, these benefits both the customer and the developer. However, if problems arise then there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

There are only a small number of experience reports on using agile methods for software maintenance (Poole and Huisman, 2001). There are two questions that should be considered when considering agile methods and maintenance:

1. Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
2. Can agile methods be used effectively for evolving a system in response to customer change requests?

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is often not kept up to date and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of

time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. Agile practices therefore emphasize the importance of writing well-structured code and investing effort in code improvement. Therefore, the lack of documentation should not be a problem in maintaining systems developed using an agile approach. However, my experience of system maintenance suggests that the key document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. In this respect, the use of agile methods is likely to make subsequent system maintenance more difficult and expensive. Agile practices, used in the maintenance process itself, are likely to be effective, whether or not an agile approach has been used for system development. Incremental delivery, design for change and maintaining simplicity all make sense when software is being changed. In fact, you can think of an agile development process as a process of software evolution.

11.9 Extreme programming

Extreme programming (XP) is perhaps the best known and most widely used of the agile methods. The name was coined by Beck (2000) because the approach was developed by pushing recognized good practice, such as iterative development, to ‘extreme’ levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated and tested in a day.

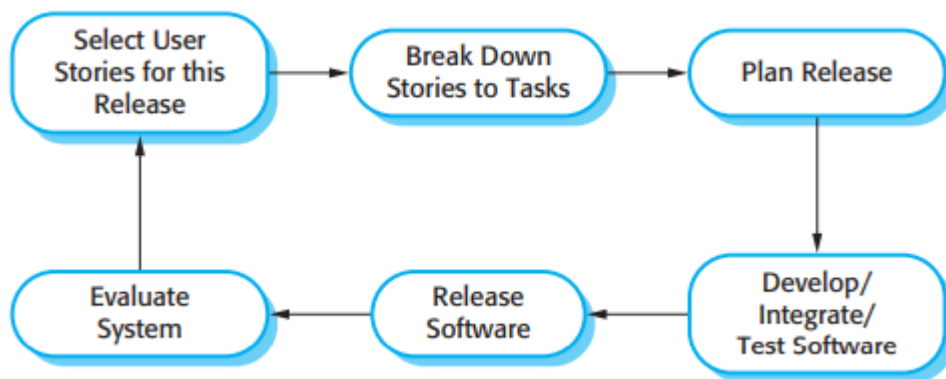


Figure 11.19 The extreme programming release cycle

In extreme programming, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short

time gap between releases of the system. Figure 11.19 illustrates the XP process to produce an increment of the system that is being developed.

Extreme programming involves a number of practices, summarized in Figure 11.20, which reflect the principles of agile methods:

Principle or practice	Description
Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development 'Tasks'.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Figure 11.20 Extreme programming practices

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.
4. Change is embraced through regular system releases to customers, test-first

development, refactoring to avoid code degeneration, and continuous integration of new functionality.

5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In an XP process, customers are intimately involved in specifying and prioritizing system requirements. The requirements are not specified as lists of required system functions. Rather, the system customer is part of the development team and discusses scenarios with other team members. Together, they develop a ‘story card’ that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the mental health care patient management system is shown in Figure 11.21.

Prescribing Medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select current medication, ‘new medication’ or ‘formulary’.

If she selects ‘current medication’, the system asks her to check the dose. If she wants to change the dose, she enters the dose and then confirms the prescription.

If she chooses ‘new medication’, the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose and then confirms the prescription.

If she chooses ‘formulary’, the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose and then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn’t, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks ‘OK’ or ‘Change’. If she clicks ‘OK’, the prescription is recorded on the audit database. If she clicks on ‘Change’, she reenters the ‘Prescribing medication’ process.

Figure 11.21 A ‘prescribing medication’ story.

This is a short description of a scenario for prescribing medication for a patient. The story cards are the main inputs to the XP planning process or the ‘planning game’. Once the story cards have been developed, the development team breaks these down into tasks (Figure 11.22) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made

available to the customer. Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

Sometimes, during the planning game, questions that cannot be easily answered come to light and additional work is required to explore possible solutions. The team may carry out some prototyping or trial development to understand the problem and solution. In XP terms, this is a ‘spike’, an increment where no programming is done. There may also be ‘spikes’ to design the system architecture or to develop system documentation. Extreme programming takes an ‘extreme’ approach to incremental development. New versions of the software may be built several times per day and releases are delivered to customers roughly every two weeks. Release deadlines are never slipped; if there are development problems, the customer is consulted, and functionality is removed from the planned release.

When a programmer builds the system to create a new version, he or she must run all existing automated tests as well as the tests for the new functionality. The new build of the software is accepted only if all tests execute successfully. This then becomes the basis for the next iteration of the system. A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn’t worth taking time to add generality to a program to cope with change. The changes anticipated often never materialize and completely different change requests may actually be made. Therefore, the XP approach accepts that changes will happen and reorganize the software when these changes actually occur.

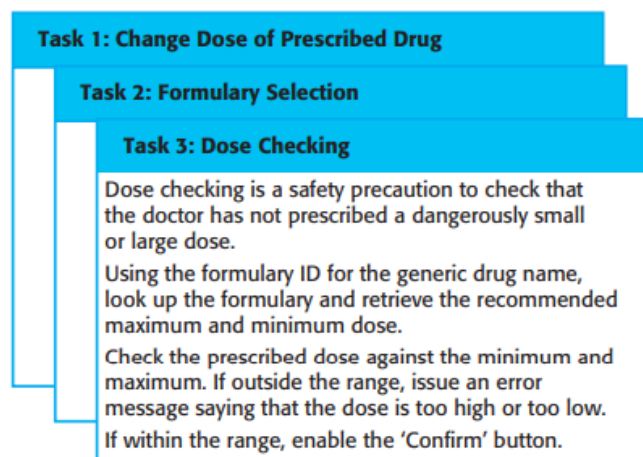


Figure 11.22 Examples of task cards for prescribing medication

A general problem with incremental development is that it tends to degrade the software structure, so changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system. Extreme programming tackles this problem by suggesting that the software should be constantly refactored. This means that the programming team look for possible improvements to the software and implement them immediately. When a team member sees code that can be improved, they make these improvements even in situations where there is no immediate need for them. Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of code with calls to methods defined in a program library. Program development environments, such as Eclipse (Carlson, 2005), include tools for refactoring which simplify the process of finding dependencies between code sections and making global code modifications. In principle then, the software should always be easy to understand and change as new stories are implemented.

In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require the architecture of the system to be modified. In practice, many companies that have adopted XP do not use all of the extreme programming practices listed in Figure 11.20. They pick and choose according to their local ways of working. For example, some companies find pair programming helpful; others prefer to use individual programming and reviews. To accommodate different levels of skill, some programmers don't do refactoring in parts of the system they did not develop, and conventional requirements may be used rather than user stories. However, most companies who have adopted an XP variant use small releases, test-first development, and continuous integration.

11.9.1 Testing in XP

To avoid some of the problems of testing and system validation, XP emphasizes the importance of program testing. XP includes an approach to testing that reduces the chances of introducing undiscovered errors into the current version of the system. The key features of testing in XP are:

1. Test-first development,
2. Incremental test development from scenarios,
3. User involvement in the test development and validation, and
4. Use of automated testing frameworks.

Test-first development is one of the most important innovations in XP. Instead of writing some code and then writing tests for that code, you write the tests before you write the code. This means that you can run the test as the code is being written and discover problems during development. Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. This approach can be adopted in any process in which there is a clear relationship between a system requirement and the code implementing that requirement. In XP, you can always see this link because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation. The adoption of test-first development in XP has led to more general test-driven approaches to development (Astels, 2003). I discuss these in Chapter 8. In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of ‘test-lag’. This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

User requirements in XP are expressed as scenarios or stories and the user prioritizes these for development. The development team assesses each scenario and breaks it down into tasks. For example, some of the task cards developed from the story card for prescribing medication (Figure 11.21) are shown in Figure 11.22. Each task generates one or more-unit tests that check the implementation described in that task. Figure 11.23 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

Test 4: Dose Checking
Input: 1. A number in mg representing a single dose of the drug. 2. A number representing the number of single doses per day.
Tests: 1. Test for inputs where the single dose is correct but the frequency is too high. 2. Test for inputs where the single dose is too high and too low. 3. Test for inputs where the single dose $\times$ frequency is too high and too low. 4. Test for inputs where single dose $\times$ frequency is in the permitted range.
Output: OK or error message indicating that the dose is outside the safe range.

Figure 11.23 Test case description for dose checking

In XP, acceptance testing, like development, is incremental. The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs. For the story in Figure 11.21, the acceptance test would involve scenarios were

- a) the dose of a drug was changed,
- b) a new drug was selected, and
- c) the formulary was used to find a drug. In practice, a series of acceptance tests rather than a single test are normally required.

Relying on the customer to support acceptance test development is sometimes a major difficulty in the XP testing process. People adopting the customer role have very limited available time and may not be able to work full-time with the development team. The customer may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process. Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing components should be stand alone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Massol and Husted, 2003) is a widely used example of an automated testing framework. As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually results in a large number of tests being written and executed. However, this approach does not necessarily lead to thorough program testing. There are three reasons for this:

1. Programmers prefer programming to testing and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the ‘display logic’ and workflow between screens.
3. It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so remain untested.

Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests written after development, then undetected bugs may be delivered in the system release.

11.9.2 Pair programming

Another innovative practice that has been introduced in XP is that programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. However, the same pairs do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process. The use of pair programming has a number of advantages:

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg's (1971) idea of egoless programming where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (covered in Chapter 24) are very successful in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Although pair programming is a less formal process that probably doesn't find as many errors as code inspections, it is a much cheaper inspection process than formal program inspections.
3. It helps support refactoring, which is a process of software improvement. The difficulty of implementing this in a normal development environment is that effort in refactoring is expended for long-term benefit. An individual who practices refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring, so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as two individuals working alone. There have been various studies of the productivity of paid programmers with mixed results. Using student volunteers,

However, more experienced programmers found that there was a significant loss of productivity compared with two programmers working alone. There were some

quality benefits, but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

11.10 Rapid Application Development

Although agile methods as an approach to iterative development have received a great deal of attention in the last few years, business systems have been developed iteratively for many years using rapid application development techniques. Rapid application development (RAD) techniques evolved from so-called fourth-generation languages in the 1980s and are used for developing applications that are data-intensive. Consequently, they are usually organized as a set of tools that allow data to be created, searched, displayed and presented in reports. Figure 11.24 illustrates a typical organization for a RAD system.

The tools that are included in a RAD environment are:

1. A database programming language that embeds knowledge of the database structure; and includes fundamental database manipulation operations. SQL is the standard database programming language. The SQL commands may be input directly or generated automatically from forms filled in by the end-user.
2. An interface generator, which is used to create forms for data input and display.
3. links to office applications such as a spreadsheet for the analysis and manipulation of numeric information or a word processor for report template creation.
4. A report generator, which is used to define and create reports from information in the database.

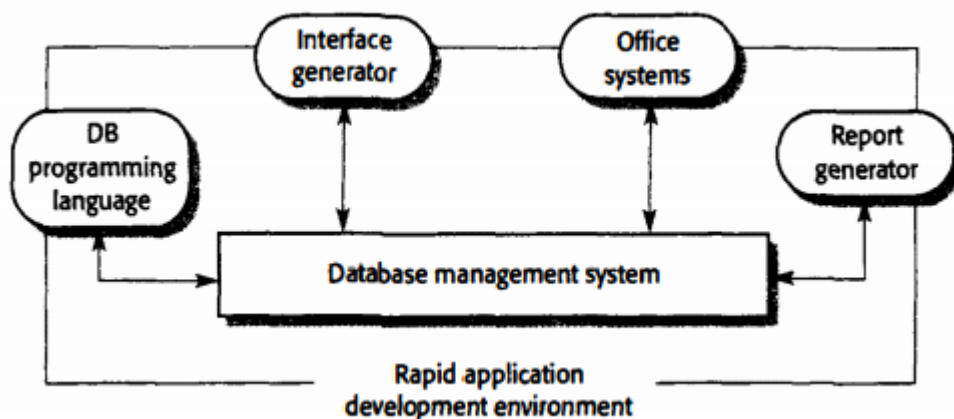


Figure 11.24 A rapid application development environment

Standard forms are used for input and output. RAD systems are geared towards producing interactive applications that rely on abstracting information from an organizational database, presenting it to end-users on their terminal or workstation, and updating the database with changes made by users. Many business applications rely on structured forms for input and output, so RAD environments provide powerful facilities for screen definition and report generation. Screens are often defined as a series of linked forms (in one application we studied, there were 137 form definitions) so the screen generation system must provide for:

1. Interactive Jann definition where the developer defines the fields to be displayed and how these are to be organized.
2. Form linking where the developer can specify that particular inputs cause further forms to be displayed.
3. Field verification where the developer defines allowed ranges for values input to form fields.

All RAD environments now support the development of database interfaces based on web browsers. These allow the database to be accessed from anywhere with a valid Internet connection. This reduces training and software costs and allows external users to have access to a database. However, the inherent limitations of web browsers and Internet protocols mean that this approach may be unsuitable for systems where very fast, interactive responses are required. Most RAD systems now also include visual programming tools that allow the system to be developed interactively. Rather than write a sequential program, the system developer manipulates graphical icons representing functions, data or user interface components, and associates processing scripts with these icons. An executable program is generated automatically from the visual representation of the system.

11.11 Software Prototyping

Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development. It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.

What is Software Prototyping?

Prototype is a working model of software with some limited functionality. The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.

Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Following is the stepwise approach to design a software prototype:

- **Basic Requirement Identification:** This step involves understanding the very basics product requirements especially in terms of user interface. The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.
- **Developing the initial Prototype:** The initial Prototype is developed in this stage, where the very basic requirements are showcased, and user interfaces are provided. These features may not exactly work in the same manner internally in the actual software developed and the workarounds are used to give the same look and feel to the customer in the prototype developed.
- **Review of the Prototype:** The prototype developed is then presented to the customer and the other important stakeholders in the project. The feedback is collected in an organized manner and used for further enhancements in the product under development.
- **Revise and enhance the Prototype:** The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like , time and budget constraints and technical feasibility of actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until customer expectations are met.

Prototypes can have horizontal or vertical dimensions. Horizontal prototype displays the user interface for the product and gives a broader view of the entire system, without concentrating on internal functions. A vertical prototype on the other side is a detailed elaboration of a specific function or a sub system in the product. The purpose of both horizontal and vertical prototype is different. Horizontal prototypes are used to get more information on the user interface level and the business requirements. It can even be presented in the sales demos to get business in the market. Vertical prototypes are technical in nature and are used to get details of the exact functioning of the sub systems. For example, database requirements, interaction and data processing loads in a given sub system.

Software Prototyping Types

There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely:

1. **Throwaway/Rapid Prototyping:** Throwaway prototyping is also called as rapid or close ended prototyping. This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype. Once the actual requirements are understood, the prototype is discarded, and the actual system is developed with a much clear understanding of user requirements.
2. **Evolutionary Prototyping:** Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning. The prototype developed forms the heart of the future prototypes on top of which the entire system is built. Using evolutionary prototyping only well understood requirements are included in the prototype and the requirements are added as and when they are understood.
3. **Incremental Prototyping:** Incremental prototyping refers to building multiple functional prototypes of the various sub systems and then integrating all the available prototypes to form a complete system.
4. **Extreme Prototyping :** Extreme prototyping is used in the web development domain. It consists of three sequential phases. First, a basic prototype with all the existing pages is presented in the html format. Then the data processing is simulated using a prototype services layer. Finally, the services are implemented and integrated to the final prototype. This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.

Software Prototyping Application

Software Prototyping is most useful in development of systems having high level of user interactions such as online systems. Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed. Software that involves too much of data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping. Prototype development could be an extra overhead in such projects and may need lot of extra efforts.

11.12 Summary:

- User interface principles covering user familiarity, consistency, minimal surprise, recoverability, user guidance and user diversity help guide the design of user interfaces. Styles of interaction with a software system include

direct manipulation, menu systems, form fill-in, command languages and natural language.

- Graphical information display should be used when it is intended to present trends and approximate values. Digital display should only be used when precision is required. Color should be used sparingly and consistently in user interfaces. Designers should take account of the fact that a significant number of people are color-blind. The user interface design process includes sub-processes concerned with user analysis, interface prototyping and interface evaluation.
- The aim of user analysis is to sensitive designers to the ways in which users actually work. You should use different techniques-task analysis, interviewing and observation-during user analysis.
- User interface prototype development should be a staged process with early prototypes based on paper versions of the interface that, after initial evaluation and feedback, are used as a basis for automated prototypes. The goals of user interface evaluation are to obtain feedback on how a UI design can be improved and to a user whether an interface meets its usability requirements.
- Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads, and producing high-quality code. They involve the customer directly in the development process.
- Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement, and customer participation in the development team.
- The Software Prototyping refers to building software application prototypes which display the functionality of the product under development but may not actually hold the exact logic of the original software.

11.13 Self-Assessment Questions:

- Why do we need UI design?
- Explain the design issues
- What is the design process?
- Write a short note on:

- a) User Analysis
 - b) RAD
 - c) Testing in XP
 - d) Interface Evaluation
- What is Software Prototyping?

11.14 List of References:

- Software Engineering, Ian Somerville, 8th edition, Pearson Education
- Software Engineering, Pankaj Jalote
- Software Engineering, A practitioner's approach, Roger Pressman, Tata McGraw-Hill

PROJECT MANAGEMENT

Unit Structure

12.0 Objectives

12.1 Introduction - Project Management

12.2 Software Project Management

12.2.1 Definition

12.2.2 Software Management Dissimilarities

12.3 Management activities

12.4 Project Planning

12.4.1 Planning stages

12.4.2 Proposal planning

12.4.3 Project plans

12.4.4 The planning process

12.5 Project Scheduling

12.5.1 Project scheduling activities

12.5.2 Milestones and deliverables

12.5.3 The project scheduling process

12.5.4 Scheduling problems

12.5.5 Schedule representation

12.5.6 Activity bar chart

12.5.7 Staff allocation chart

12.5.8 Agile planning

12.5.8.1 Agile planning stages

12.5.8.2 Planning in XP

12.5.8.3 Story-based planning

12.6 Risk Management

12.6.1 Definition

12.6.2 Examples of common project, product, and business risks

12.6.3 The risk management process

12.6.4 Risk identification

12.6.5 Examples of different risk types

12.6.6 Risk analysis

12.6.7 Risk types and examples

12.6.8 Risk planning

12.6.9 Strategies to help manage risk

12.6.10 Risk monitoring

12.6.11 Risk indicators

12.7 Let's Sum up

12.8 References

12.9 Unit End Exercises

12.0 Objectives

This Chapter would make you understand the following concepts:

- What is Project Management
- Management process of Software Project
- Management activities related with the software Project
- Stages, Planning and Process of Project Planning
- Activities, Process, Project Scheduling
- Risk Management

12.1 Introduction – Project Management

Project management is the use of specific knowledge, skills, tools and techniques to deliver something of value to people. The development of software for an improved business process, the construction of a building, the relief effort after a natural disaster, the expansion of sales into a new geographic market—these are all examples of projects.

12.2 Software Project Management

12.2.1 Definition:

Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organizations developing and procuring the software.

- Project management is needed because software development is always subject to budget and schedule constraints that are set by the organizations developing the software.
- Deliver the software to the customer at the agreed time.
- Keep overall costs within budget.
- Deliver software that meets the customer's expectations.
- Maintain a happy and well-functioning development team.

12.2.2 Software Management Dissimilarities

- **The product is intangible.**

Software cannot be seen or touched. Software project managers cannot see progress by simply looking at the artefact that is being constructed.

- **Many software projects are 'one-off' projects.**

Large software projects are usually different in some ways from previous projects. Even managers who have lots of previous experience may find it difficult to anticipate problems.

- **Software processes are variable and organization specific.**

We still cannot reliably predict when a particular software process is likely to lead to development problems.

12.3 Management activities

- **Project planning**

Project managers are responsible for planning. Estimating and scheduling project development and assigning people to tasks.

- **Reporting**

Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.

- **Risk management**

Project managers assess the risks that may affect a project, monitor these risks and take action when problems arise.

- **People management**

Project managers have to choose people for their team and establish ways of working that leads to effective team performance

- **Proposal writing**

The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out.

12.4 Project planning

Project planning involves breaking down the work into parts and assign these to project team members, anticipate problems that might arise and prepare tentative solutions to those problems.

The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

12.4.1 Planning stages

At the proposal stage, when you are bidding for a contract to develop or provide a software system.

During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc.

Periodically throughout the project, when you modify your plan in the light of experience gained and information from monitoring the progress of the work.

12.4.2 Proposal planning

Planning may be necessary with only outline software requirements.

The aim of planning at this stage is to provide information that will be used in setting a price for the system to customers.

12.4.3 Project plans

In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work.

Plan sections

- Introduction
- Project organization
- Risk analysis
- Hardware and software resource requirements
- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

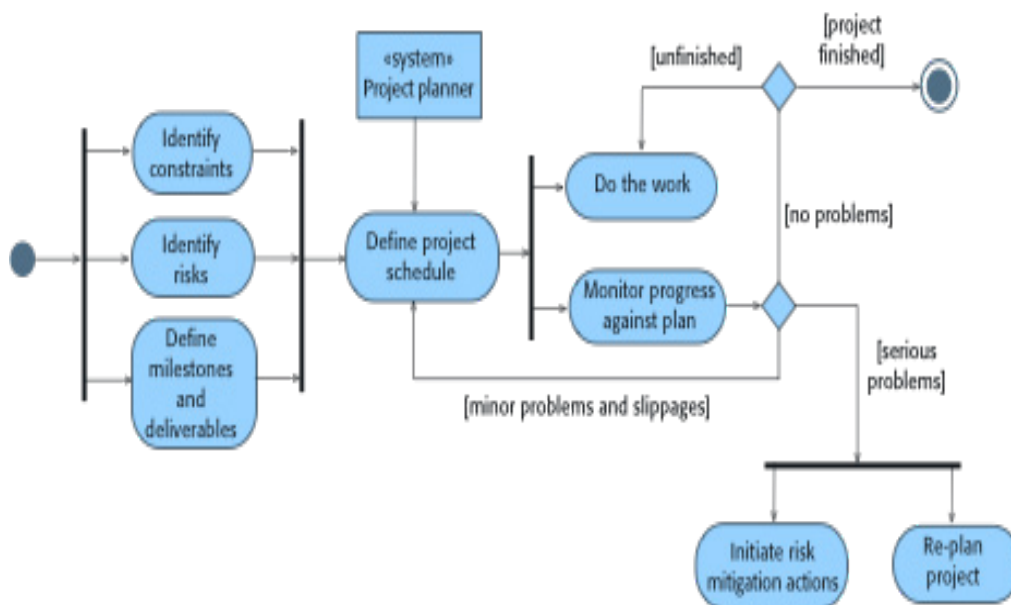
Project plan supplements

PLAN	DESCRIPTION
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

12.4.4 The planning process

- Project planning is an iterative process that starts when you create an initial project plan during the project startup phase.
- Plan changes are inevitable.
 - As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule and risk changes.
 - Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be re-planned.

The project planning process



12.5 PROJECT SCHEDULING

- Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- You estimate the calendar time needed to complete each task, the effort required and who will work on the tasks that have been identified.
- You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be.

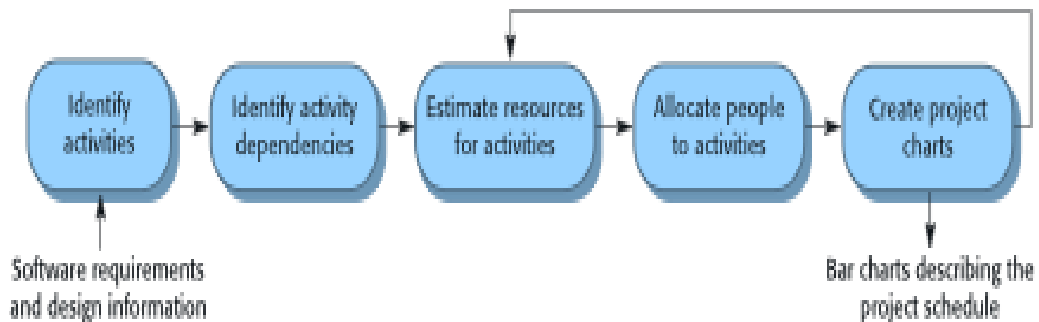
12.5.1 Project scheduling activities

- Split project into tasks and estimate time and resources required to complete each task.
- Organize tasks concurrently to make optimal use of workforce.
- Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- Dependent on project managers intuition and experience.

12.5.2 Milestones and deliverables

- Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing.
- Deliverables are work products that are delivered to the customer, e.g. a requirements document for the system.

12.5.3 The project scheduling process



12.5.4 Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- Productivity is not proportional to the number of people working on a task.
- Adding people to a late project makes it later because of communication overheads.
- The unexpected always happens. Always allow contingency in planning.

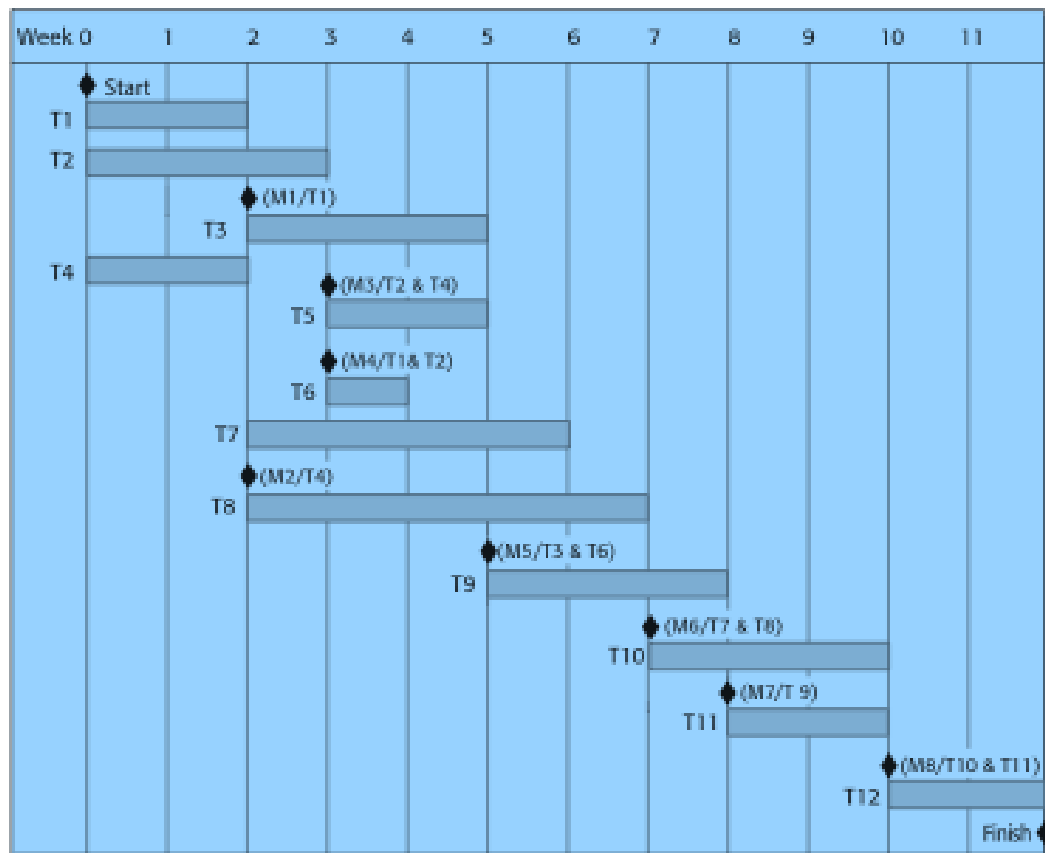
12.5.5 Schedule representation

- Graphical notations are normally used to illustrate the project schedule.
- These show the project breakdown into tasks. Tasks should not be too small. They should take about a week or two.
- Bar charts are the most commonly used representation for project schedules. They show the schedule as activities or resources against time.

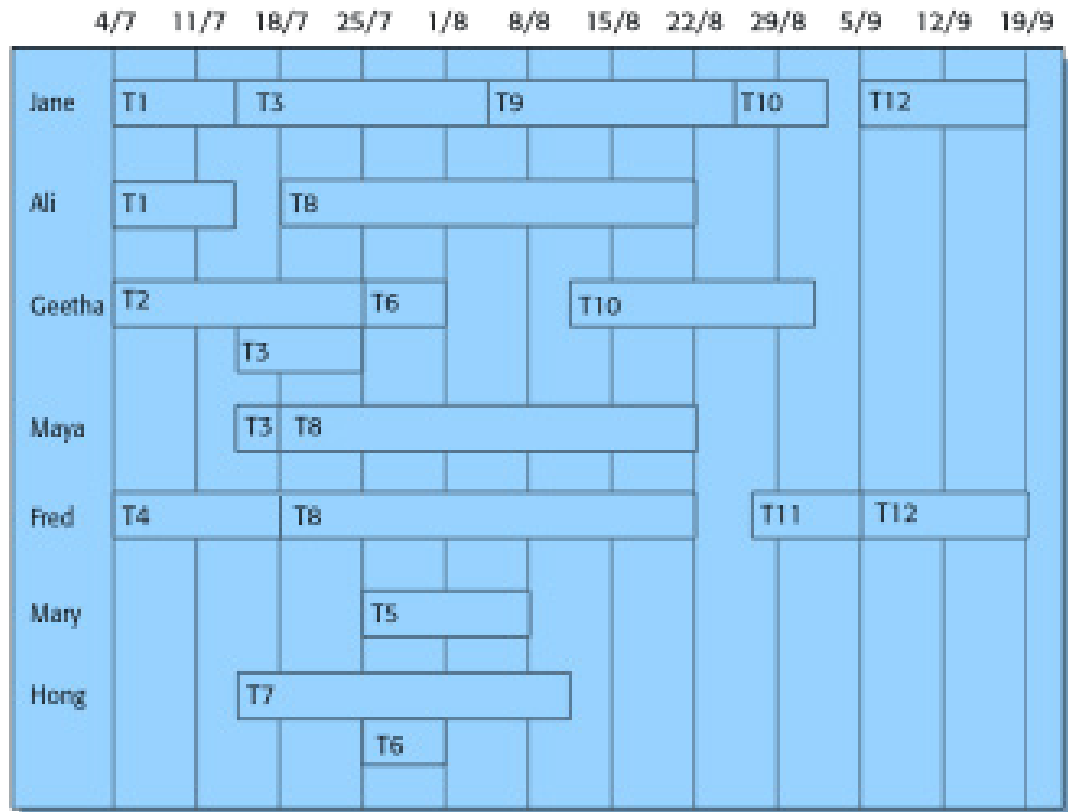
Tasks, durations, and dependencies

Task	Effort (days)	(person	Duration (days)	Dependencies
T1	15		10	
T2	8		15	
T3	20		15	T1 (M1)
T4	5		10	
T5	5		10	T2, T4 (M3)
T6	10		5	T1, T2 (M4)
T7	25		20	T1 (M1)
T8	75		25	T4 (M2)
T9	10		15	T3, T6 (M5)
T10	20		15	T7, T8 (M6)
T11	10		10	T9 (M7)
T12	20		10	T10, T11 (M8)

12.5.6 Activity bar chart



12.5.7 Staff allocation chart



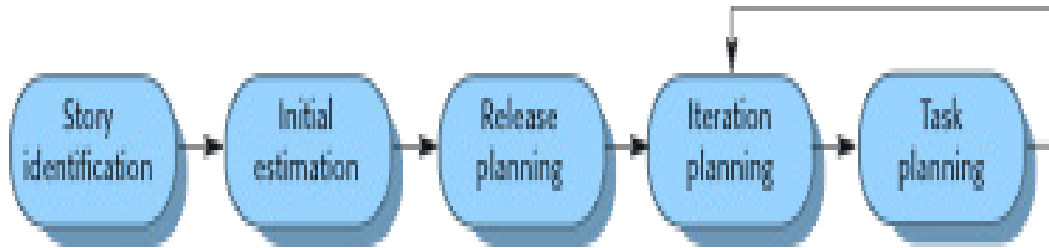
12.5.8 Agile planning

- Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments.
- Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development.
- The decision on what to include in an increment depends on progress and on the customer's priorities.
- The customer's priorities and requirements change so it makes sense to have a flexible plan that can accommodate these changes.

12.5.8.1 Agile planning stages

- Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
- Iteration planning, which has a shorter term outlook, and focuses on planning the next increment of a system. This is typically 2-4 weeks of work for the team.

12.5.8.2 Planning in XP



12.5.8.3 Story-based planning

- The system specification in XP is based on user stories that reflect the features that should be included in the system.
- The project team read and discuss the stories and rank them in order of the amount of time they think it will take to implement the story.
- Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented.
- Stories to be implemented in each iteration are chosen, with the number of stories reflecting the time to deliver an iteration (usually 2 or 3 weeks).

12.6 RISK MANAGEMENT

12.6.1 Definition:

Risk management is concerned with identifying risks and drawing up plans to minimize their effect on a project.

- A risk is a probability that some adverse circumstance will occur
- Project risks affect schedule or resources;
- Product risks affect the quality or performance of the software being developed;
- Business risks affect the organization developing or procuring the software.

12.6.2 Examples of common project, product, and business risks

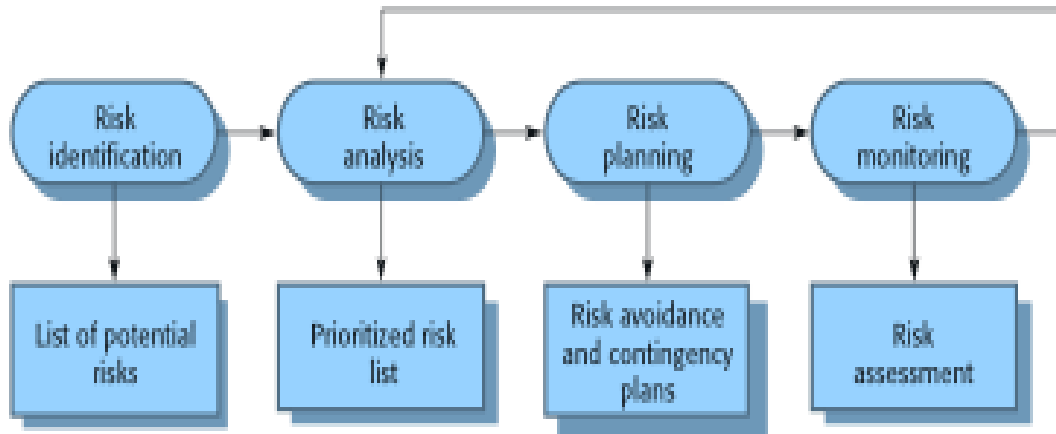
RISK	AFFECTS	DESCRIPTION
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.

Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

12.6.3 The risk management process

- Risk identification
 - Identify project, product and business risks;
- Risk analysis
 - Assess the likelihood and consequences of these risks;
- Risk planning
 - Draw up plans to avoid or minimise the effects of the risk;
- Risk monitoring
 - Monitor the risks throughout the project;

The risk management process



12.6.4 Risk identification

- May be a team activities or based on the individual project manager's experience.
- A checklist of common risks may be used to identify risks in a project
 - Technology risks.
 - People risks.
 - Organizational risks.
 - Requirements risks.
 - Estimation risks.

12.6.5 Examples of different risk types

RISK TYPE	POSSIBLE RISKS
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)

Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)

12.6.6 Risk analysis

- * Assess probability and seriousness of each risk.
- * Probability may be very low, low, moderate, high or very high.
- * Risk consequences might be catastrophic, serious, tolerable or insignificant.

12.6.7 Risk types and examples

RISK	PROBABILITY	EFFECTS
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	Catastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious

The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant

12.6.8 Risk planning

Consider each risk and develop a strategy to manage that risk.

- Avoidance strategies
 - The probability that the risk will arise is reduced
- Minimizations strategies
 - The impact of the risk on the project or product will be reduced
- Contingency plans
 - If the risk arises, contingency plans are plans to deal with that risk

12.6.9 Strategies to help manage risk

RISK	STRATEGY
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

12.6.10 Risk monitoring

- Assess each identified risks regularly to decide whether or not it is becoming less or more probable.
- Also assess whether the effects of the risk have changed.
- Each key risk should be discussed at management progress meetings.

12.6.11 Risk indicators

RISK TYPE	POTENTIAL INDICATORS
Technology	Late delivery of hardware or support software; many reported technology problems.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Organizational	Organizational gossip; lack of action by senior management.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.
Requirements	Many requirements change requests; customer complaints.
Estimation	Failure to meet agreed schedule; failure to clear reported defects.

12.7 Let's Sum up

- We will have a clear idea about Management of Software Project, Management activities related with the software Project , Stages , Planning and Process of software Project Planning Activities, Process, Project Scheduling and Risk Management

12.8 References

- <https://www.pmi.org/about/learn-about-pmi/what-is-project-management#:~:text=Project%20management%20is%20the%20use,something%20of%20value%20to%20people.&text=Each%20project%20is%20unique%20and,once%20the%20goal%20is%20achieved.>
- projectmanagement.com/default.cfm#_=_
- <https://www.javatpoint.com/software-project-management-activities>
- https://www.tutorialspoint.com/software_engineering
- http://moodle.autolab.uni-pannon.hu/Mecha_tananyag/szoftverfejlesztési_folyamatok_angol/ch11.html

- <https://www.geeksforgeeks.org/software-engineering-project-management-process/>
- <http://www.cs.ox.ac.uk/people/michael.wooldridge/teaching/soft-eng/lect05.pdf>
- <https://ppqc.net/assets/Example%20Project%20Planning%20Process.pdf>
- <https://ecomputernotes.com/software-engineering/project-planning>
- <https://www.sinnaps.com/en/project-management-blog/project-planning-in-software-engineering>
- <https://cs.ccsu.edu/~stan/classes/CS530/Notes18/23-ProjectPlanning.html>
- https://ccelms.ap.gov.in/adminassets/docs/24112020155304-SE-KNOW_MORE-1-18.pdf
- <https://www.wrike.com/project-management-guide/faq/what-is-scheduling-in-project-management/>
- <https://tutorialsinhand.com/tutorials/software-engineering-tutorial/software-project-management/software-project-scheduling.aspx>
- <https://kissflow.com/project/basics-of-project-scheduling/>
- <https://www.castsoftware.com/research-labs/risk-management-in-software-development-and-software-engineering-projects>
- <https://www.guru99.com/risk-analysis-project-management.html>
- <https://www.unf.edu/~ncoulter/cen6070/handouts/ManagingRisk.pdf>

12.9 Unit End Exercises

- Take a Real- time Project and prepare the chart of Management activities related with the software Project.
- Take a Real- time Project and list out the stages of software Project.
- Take a Real- time Project and write up the Scheduling for the Project.
- Take a Real- time Project prepare the Risk factors of the Software Project.

QUALITY MANAGEMENT

Unit Structure

- 16.1 Quality Management
 - 16.1.1 Quality Factors
 - 16.1.2 Quality Management and Software development
- 16.2 Software Quality
 - 16.2.1 Software Quality attribute
- 16.3 Quality Assurance and software Standards
 - 16.3.1 ISO 9001 standards framework
- 16.4 Review and inspection
 - 16.4.1 The review process
- 16.5 Soft-ware measurement and metrics
 - 16.5.1 Product metrics
 - 16.5.2 Software component analysis

13.0 Objectives

- Understand the software quality and measurement of the product/software.
- Software planning and process is important. Software.
- Software planning can decrease the quality of the product.
- Understand how measurement is help full.

13.1 Quality Management

The quality is the capacity of commodity to satisfy human wants and needs better the quality higher the cost half the product which means as you satisfy all the requirement of the customer which are specified by individual customer and if those requirements are specifications providing output according to the requirement and giving us high quality output with maximum consistency then we can called any product as a quality product

Software quality management for software systems has three principal concerns:

1. The organizational level:

Quality management is concerned with establishing a framework of organizational processes and standards that will lead to high quality software. This means that the quality management team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.

2. The project level:

Quality management involves the application of specific quality processes, checking that these planned processes have been followed, and ensuring that the project outputs are conformant with the standards that are applicable to that project.

3. Quality management:

at the project level is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

13.1.1 There are several quality factors which define a quality product as follows

Portability:

Software must be platform independent. It must be easily installed in any of the operating system, or it can be installed in any system environment, in any machine with other software. It can be stored in any of the device (CD, Pen-drive) while moving the software from one point to another point for installation

Usability:

Every product has good usability, but while using the product the working of the product needs to be understood by the end user or by the owner of that particular product. Understanding the product process is most important part in usability so that any expert or novice user can easily understand and use the product.

Re usability:

The software product must provide re-usability, as it can be used here in other modules while developing a new module, such of software module here need to be used all developed as it will save more time money on implementing different projects or modules.

Correct Ness:

The requirements which are provided by the customer must be implemented in the given project if all the requirements which are specified and working properly and giving quality output and consistency then we can measure the correct Ness of the product.

Maintainability:

Maintainability is nothing but we can do the changes into the system as per our requirement, whenever we want but it could not affect any part of the designing.

If any error occurs, then we can able to modify or remove error from the software. As per requirement we can add new functionalities in the system as well.

Software quality management provide independent check on software development process, for small system development, software quality management hey, it is done with the help of a proper communication between team members which include external customer as well who provide views on the project which is developed or in developing mode.

For large system we will do software quality management with respect to following points

Quality assurance:

Any product needs to assure the customer with its quality output, as it aims in building the organizational procedures and standards. Procedure/Output assure the quality of the product the market branding for the product as well As for organization may get increased.

Quality planning:

Better the plain best the product hey so that you have to select best planning procedure to build any product selecting appropriate procedure and proper planning increase the quality output of the product.

Quality control:

Quality of the product hey need to be controlled as it need to be consistent throughout the procedure. So, we need to implement a product or module by following quality procedures and standards.

13.1.2 Quality Management and Software development

- Quality management provides an independent check on the software development process. The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals.

- The QA team should be independent from the development team so that they can take an objective view of the software.
- This allows them to report on software quality without being influenced by software development issues.



1. From the above diagram we can see that quality management provides an independent check on the software development process.
 2. The quality management process checks the project deliverable to ensure that they are consistent with organizational standards and goals. This allows them to report on software quality without being influenced by software development issues.
 3. software development team is independent from quality management team.
- Quality planning is the process of developing a quality plan for a project.
 - The quality plan should set out the desired software qualities and describe how these are to be assessed.
 - It therefore defines what 'high-quality software actually means for a particular system. Without this definition, engineers may make different sometimes conflicting assumptions about which product attributes reflect the most important quality characteristics.
 - Formalized quality planning is an integral part of plan-based development processes.
 - Agile methods, however, adopt a less formal approach to quality management.

An out-line structure for a quality plan. This includes:

1. Product introduction A description of the product, its intended market, and the quality expectations for the product.

2. Product plans the critical release dates and responsibilities for the product along with plans for distribution and product servicing.
3. Process descriptions the development and service processes and standards that should be used for product development and management.
4. Quality goals the quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. Risks and risk management the key risks that might affect product quality and the actions to be taken to address these risks.

13.2 Software Quality

The fundamentals of quality management were established by manufacturing industry in a drive to improve the quality of the products that were being made.

Software quality is not directly comparable with quality in manufacturing. The idea of tolerances is not applicable to digital systems, and, for the following reasons, it may be impossible to come to an objective conclusion about whether or not a software system meets its specification.

1. Difficult to write the complete an unambiguous software specification. hey Software developer and customer may interpret the requirements in different way.
2. specification usually integrate requirement from several classes of stakeholders. These requirements are inevitably a compromise and may not be included the requirement of all stakeholders group.
3. It is impossible to measure certain quality characteristics directly.

13.2.1 Software Quality attribute

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

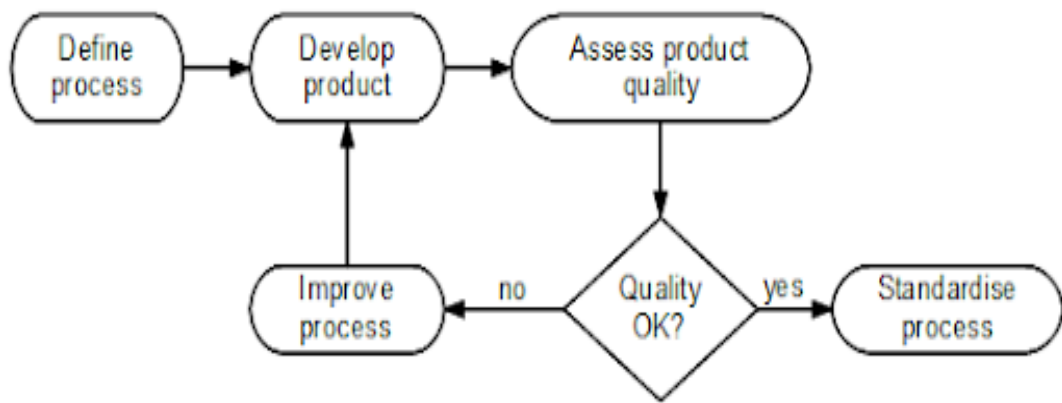


Fig: Process-based quality

- There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor.
- Once manufacturing systems are calibrated, they can be run again and again to output high-quality products.
- However, software is not manufactured-it is designed. In software development, therefore, the relationship between process quality and product quality is more complex.
- Software development is a creative rather than a mechanical process, so the influence of individual skills and experience is significant.
- External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.
- The development process used has a significant influence on the quality of the software and those good processes are more likely to lead to good quality software.
- Process quality management and improvement can lead to fewer defects in the software being developed. However, it is difficult to assess software quality attributes, such as maintainability, without using the software for a long period.
- Consequently, it is hard to tell how process characteristics influence these attributes.

13.3 Quality Assurance and software Standards

Quality assurance is a planned and systematic approach necessary to provide a high degree of confidence in the quality of a product. It provides quality assessment of the quality control activities and determines the validity of the data for identifying the quality.

The main function of quality assurance is to define the standards as:

1. Product standards
2. Process standards

1. Product standards

These apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

2. Process standards

These define the processes that should be followed during software development. They should encapsulate good development practice. Process standards may include definitions of specification, design and validation processes, process support tools, and a description of the documents that should be written during these processes.

Software standards are important for three reasons:

1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company. Building it into a standard helps the company reuse this experience and avoid previous mistakes.
2. Standards provide a framework for defining what 'quality' means in a particular setting, software quality is subjective, and by using standards you establish a basis for deciding if a required level of quality has been achieved.

Product standards	Process standards
Design review conduct	Design review form
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Of course, this depends on setting standards that reflect user expectations for software dependability, usability, and performance.

3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices. Consequently, the learning effort required when starting new work is reduced.

As there are two type pf standards there are many Product and Process standards which are depicted in table below.

To minimize dissatisfaction and to encourage buy-in to standards, quality managers who set the standards should therefore take the following steps:

1. **Involve software engineers in the selection of product standards**

If developers understand why standards have been selected, they are more likely to be com mitted to these standards. Ideally, the standards document should not just set out the standard to be followed but should also include commentary explaining why standardization decisions have been made.

2. **Review and modify standards regularly to reflect changing technologies**

Standards are expensive to develop and they tend to be enshrined in a company standards handbook. Because of the costs and discussion required, there is often a reluctance to change them. A standards handbook is essential, but it should evolve to reflect changing circumstances and technology.

3. **Provide software tools to support standards**

Developers often find standards to be a bugbear when conformance to them involves tedious manual work that could be done by a software tool. If tool support is available, very little effort is required to follow the software development standards. For example, document standards can be implemented using word processor styles.

16.3.1 The ISO 9001 standards framework

- There is an international set of standards that can be used in the development of quality management systems in all industries, called ISO 9000.
- ISO 9000 standards can be applied to a range of organizations from manufacturing through to service industries.
- ISO 9001, the most general of these standards, applies to organizations that design, develop, and maintain products, including software.
- The ISO 9001 standard is not itself a standard for software development but is a framework for developing software standards.

- It sets out general quality principles, describes quality processes in general, and lays out the organizational standards and procedures that should be defined.
- These should be documented in an organizational quality manual.
- The major revision of the ISO 9001 standard in 2000 reoriented the standard around nine core processes

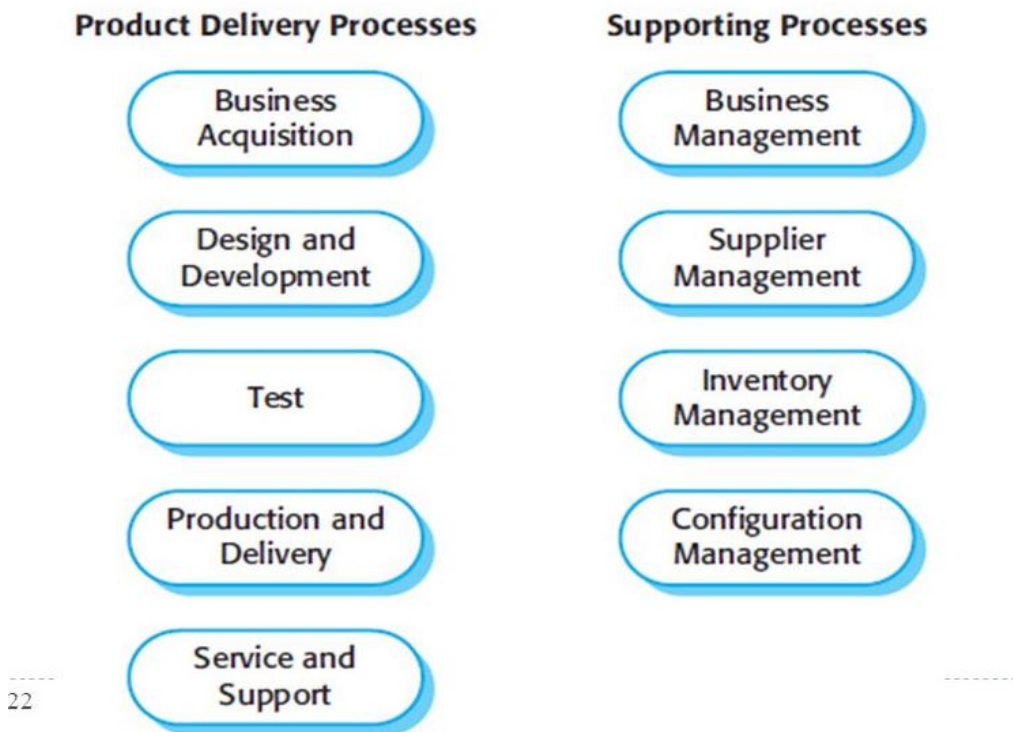


Fig: ISO 9001 core processes

- The relationships between ISO 9001, organizational quality manuals, and individual
- project quality plans are shown in ISO 9001 and quality management shown in following Figure.
- This diagram has been derived from a model given by Ince (1994), who explains how the general ISO 9001 standard can be used as a basis for software quality management processes.

The guideline steps are:

- Support the quality
- Satisfy the customer
- Establish quality policy
- Control quality system

- Provide quality resources
- Provide quality personnel
- Document the quality management system
- Perform management reviews.
- conduct quality system
- control customer processes
- control monitoring devices
- analyse quality information



Fig: ISO 9001 and quality management

Review and inspection

1. Reviews and inspections are QA activities that check the quality of project deliverables. This involves examining the software, its documentation, and records of the process to discover errors and omissions and to see if quality standards have been followed.
2. During a review, a group of people examine the software and its associated documentation, looking for potential problems and non-conformance with standards.
3. The review team makes informed judgments about the level of quality of a system or project deliverable. Project managers may then use these assessments to make planning decisions and allocate resources to the development process.

The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team.

1. Reviewing is a public process of error detection, compared with the more private component-testing process.
2. Inevitably, mistakes that are made by individuals are revealed to the whole programming team.
3. To ensure that all developers engage constructively with the review process, project managers must be sensitive to individual concerns.
4. They must develop a working culture that provides support without blame when errors are discovered.

Although a quality review provides information for management about the software being developed, quality reviews are not the same as management progress reviews.

Progress reviews take external factors into account and changed circumstances may mean that software under development is no longer required or has to be radically changed.

16.3.2 The review processes

1. Pre-review activities

These are preparatory activities that are essential for the review to be effective. Typically, pre-review activities are concerned with review planning and review preparation. Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed. During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant standards. They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.

2. The review meeting

During the review meeting, an author of the document or program being reviewed should walk through' the document with the review team. The review itself should be relatively short-two hours at most. One team member should chair the review, and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all written comments are considered. The review chair should sign a record of comments and actions agreed during the review.

3. Post-review activities

After a review meeting has finished, the issues and problems raised during the review must be addressed.

This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents.

Sometimes, the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them.

After changes have been made, the review chair may check that the review comments have all been considered.

Sometimes, a further review will be required to check that the changes made cover all the previous review comments.

13.5 Software measurement and metrics

- Software measurement is concerned with deriving a numeric value or profile for an attribute of a software component, system, or process.
- By comparing these values to each other and to the standards that apply across an organization, you may be able to draw conclusions about the quality of software, or assess the effectiveness of software processes, tools, and methods.
- The long-term goal of software measurement is to use measurement in place of reviews to make judgments about software quality.
- Using software measurement, a system could ideally be assessed using a range of metrics and, from these measurements, a value for the quality of the system could be inferred.
- If the software had reached a required quality threshold, then it could be approved without review.
- When appropriate, the measurement tools might also highlight areas of the software that could be improved. However, we are still quite a long way from this ideal situation and, there are no signs that automated quality assessment will become a reality in the foreseeable future.

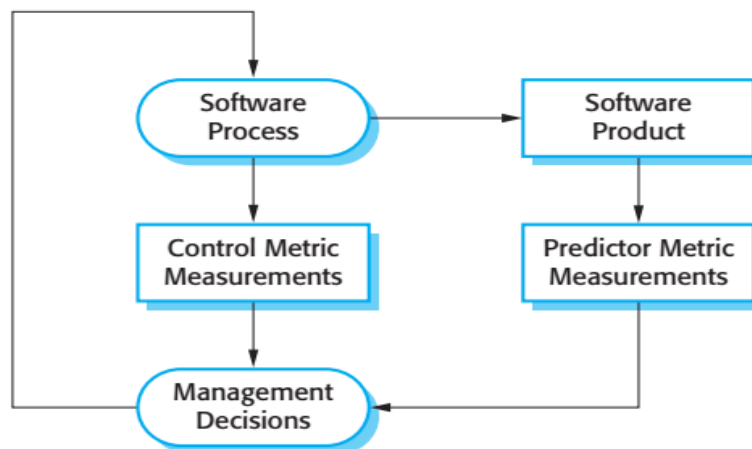


Fig: Predictor and control measurement

- A software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured.
- Examples of metrics include the size of a product in lines of code.
- Software metrics may be either control metrics or predictor metrics. As the names imply, control metrics support process management, and predictor metrics help you predict characteristics of the software.
- Control metrics are usually associated with software processes.
- Examples of control or process metrics are the average effort and the time required to repair reported defects.

There are two ways in which measurements of a software system may be used:

1. To assign a value to system quality attributes

By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.

2. To identify the system components whose quality is substandard

Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

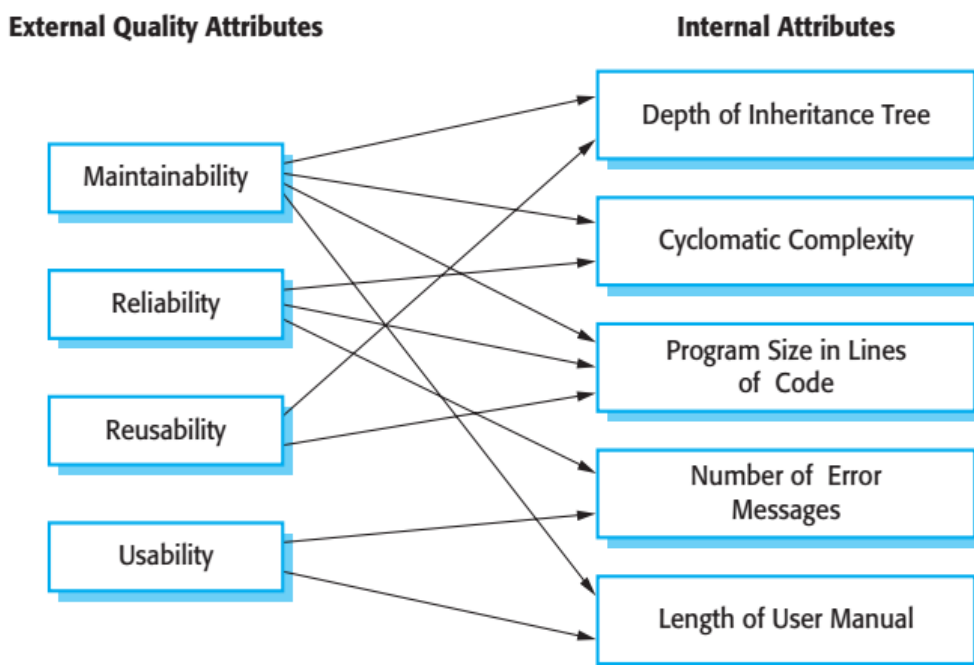


Fig: Relationships between internal and external software

The diagram suggests that there may be relationships between external and internal attributes, but it does not say how these attributes are related. If the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold.

1. **The internal attribute must be measured accurately.**

This is not always straightforward, and it may require special-purpose tools to make the measurements.

2. **A relationship must exist between the attribute**

that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related, in some way, to the value of the attribute that can be measured.

3. **This relationship between the internal and external attributes**

must be understood, validated, and expressed in terms of a formula or model. Model formulation involves identifying the functional form of the model (linear, exponential, etc.) by analysis of collected data, identifying the parameters that are to be included in the model, and calibrating these parameters using existing data.

13.5.1 Product metrics

Product metrics are predictor metrics that are used to measure internal attributes of a software system. Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class.

Product metrics fall into two classes:

1. **Dynamic metrics**, which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.
 2. **Static metrics**, which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are the code size and the average length of identifiers used.
- These types of metrics are related to different quality attributes. Dynamic metrics help to assess the efficiency and reliability of a program. Static metrics help assess the complexity, understandability, and maintainability of a software system or system components.

- There is usually a clear relationship between dynamic metrics and software quality characteristics.
- It is easy to measure the execution time required for functions and to assess the time required to start up a system.
- These relate directly to the system's efficiency. Similarly, the number of system failures and the type of failure can be logged and related directly to the reliability of the software

13.5.2 Software component analysis

A measurement process that may be part of a software quality assessment process is shown in below Figure Each system component can be analysed separately using a range of metrics. The values of these metrics may then be compared for different components and, perhaps, with historical measurement data collected on previous projects. Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

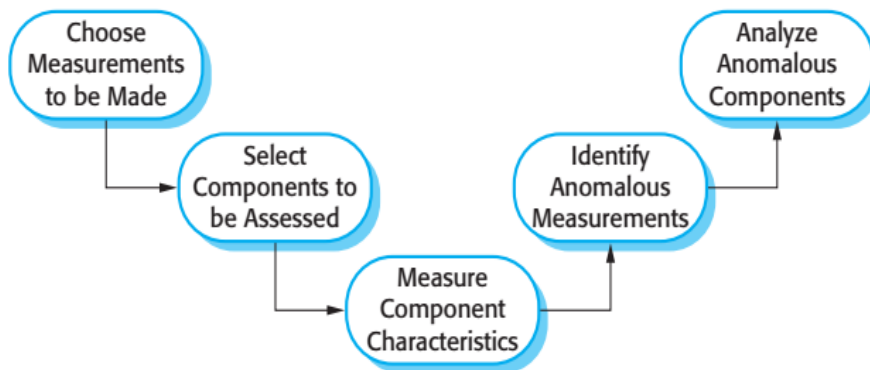


Fig: The process of product measurement

The key stages in this component measurement process are:

1. Choose measurements to be made

The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined. Measurements that are not directly relevant to these questions need not be collected.

2. Select components to be assessed

You may not need to assess metric values for all the components in a software system. Sometimes, you can select a representative selection of components for measurement, allowing you to make an overall assessment of system

quality. At other times, you may wish to focus on the core components of the system that are in almost constant use. The quality of these components is more important than the quality of components that are only rarely used.

3. Measure component characteristics

The selected components are measured, and the associated metric values computed. This normally involves processing the component representation (design, code, etc.) using an automated data collection tool. This tool may be specially written or may be a feature of design tools that are already in use.

4. Identify anomalous measurements

After the component measurements have been made, you then compare them with each other and to previous measurements that have been recorded in a measurement database. You should look for unusually high or low values for each metric, as these suggest that there could be problems with the component exhibiting these values.

5. Analyse anomalous components

When you have identified components that have anomalous values for your chosen metrics, you should examine them to decide whether these anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity (say) does not necessarily mean a poor-quality component. There may be some other reason for the high value, so may not mean that there are component quality problems.

Graded Question

1. Explain the process of Software Quality Management
2. Explain quality assurance and standards
3. Explain how product quality can be planned
4. Explain what is software metric and measurement?
5. What are the key component measurement process?

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

VERIFICATION AND VALIDATION

Unit Structure

- 14.0 Objective
- 14.1 Verification and Validation
- 14.2 Software Inspections
- 14.3 Automated Static Analysis
- 14.4 Verification and Formal Methods
- 14.5 Cleanroom software development
- 14.6 Summary
- 14.7 Exercise

14.0 Objectives

- The objective of this chapter is to introduce software verification and validation techniques. This chapter helps us to understand the distinctions between software verification and software validation.
- Software inspection is introduced to program inspections as a method of discovering defects in programs.
- We can understand what automated static analysis is and how it is used in verification and validation.
- Also understand how static verification is used in the development process. To describe the Cleanroom software development process

14.1 Verification and Validation

Verification and Validation is the process of investigating that a software system satisfies specifications and standards and it fulfills the required purpose.

Barry Boehm described verification and validation as the following:

Verification: *Are we building the product, right?*

Validation: *Are we building the right product?*

Verification:

- Verification is the process of checking that a software achieves its goal without any bugs.
- It is the process to ensure whether the product that is developed is right or not.

- It verifies whether the developed product fulfills the requirements that we have.
- Verification is **Static Testing**.
- Activities involved in verification:
 1. Inspections
 2. Reviews
 3. Walkthroughs
 4. Desk-checking

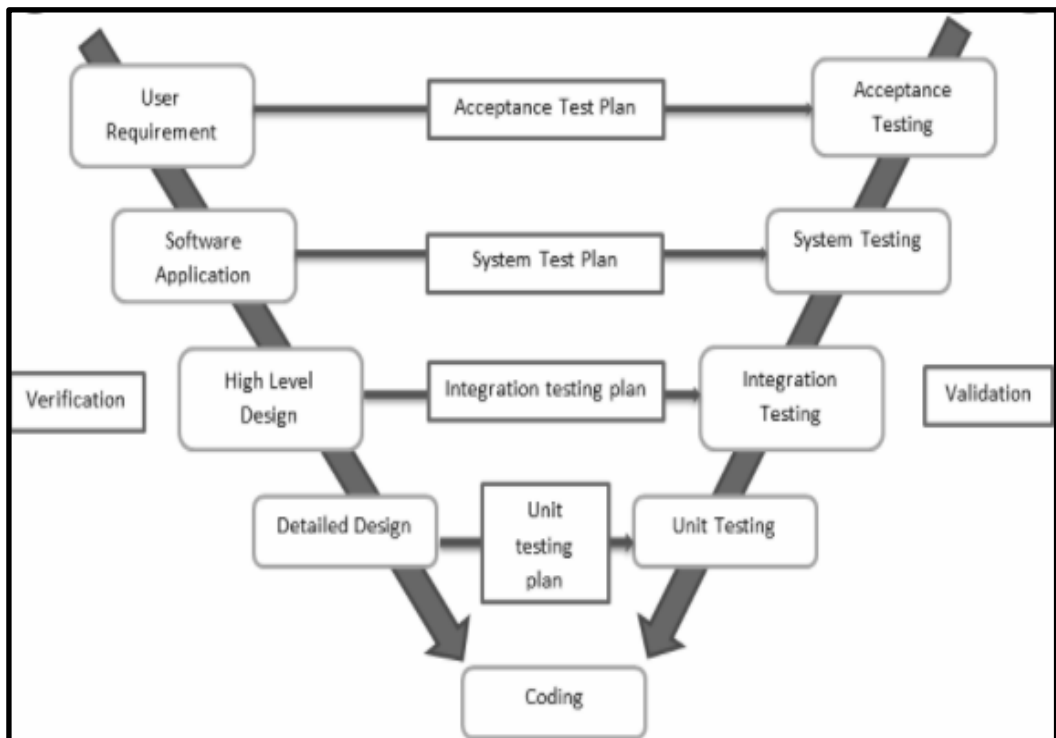
Validation:

- Validation is the process of checking whether the software product is up to the mark.
- In other words, the product has high level requirements.
- It is the process of checking the validation of product i.e., it checks what we are developing is the right product.
- It is validation of actual and expected products.
- Validation is the **Dynamic Testing**.

Verification is followed by Validation.



Diagrammatic representation of Verification and validation model:



Difference between verification and validation testing:

Verification	Validation
We check whether we are developing the right product or not.	We check whether the developed product is right.
Verification is also known as static testing .	Validation is also known as dynamic testing .
Verification includes different methods like Inspections, Reviews, and Walkthroughs.	Validation includes testing like functional testing, system testing, integration, and User acceptance testing.
It is a process of checking the work-products (not the final product) of a development cycle to decide whether the product meets the specified requirements.	It is a process of checking the software during or at the end of the development cycle to decide whether the software follow the specified business requirements.
Quality assurance comes under verification testing.	Quality control comes under validation testing.
The execution of code does not happen in the verification testing.	In validation testing, the execution of code happens.
In verification testing, we can find the bugs early in the development phase of the product.	In the validation testing, we can find those bugs, which are not caught in the verification process.
Verification testing is executed by the Quality assurance team to make sure that the product is developed according to customers' requirements.	Validation testing is executed by the testing team to test the application.
Verification is done before the validation testing.	After verification testing, validation testing takes place.
In this type of testing, we can verify that the inputs follow the outputs or not.	In this type of testing, we can validate that the user accepts the product or not.

KEY DIFFERENCE

- Verification process includes checking of documents, design, code and program whereas the Validation process includes testing and validation of the actual product.
- Verification does not involve code execution while Validation involves code execution.
- Verification uses methods like reviews, walkthroughs, inspections and desk-checking whereas Validation uses methods like black box testing, white box testing and non-functional testing.
- Verification checks whether the software confirms a specification whereas Validation checks whether the software meets the requirements and expectations.
- Verification finds the bugs early in the development cycle whereas Validation finds the bugs that verification can not catch.
- Verification process targets software architecture, design, database, etc. while the Validation process targets the actual software product.
- Verification is done by the QA team while Validation is done by the involvement of testing team with QA team.
- Verification process comes before validation whereas the Validation process comes after verification.

14.2 Software Inspections:

- Software inspection is a static V & V process in which a software system is reviewed to find errors, omissions and anomalies.
- It focuses on source code, but any readable representation of the software such as its requirements or a design model can be inspected.
- While inspecting a system, you use knowledge of the system, its application domain and the programming language or design model to discover errors.
- There are three major advantages of inspection over testing:
 1. *During testing, errors can mask (hide) other errors.*
 - ☐ Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error.
 - ☐ Because inspection is a static process, you don't have to be concerned with interactions between errors.
 - ☐ Consequently, a single inspection session can discover many errors in a system.

2. *Incomplete versions of a system can be inspected without additional costs.*

- ☐ If a program is incomplete, then you need to develop specialised test harnesses to test the parts that are available.
 - ☐ This obviously adds to the system development costs.
 - ☐ 3. *Searching for program defects, an inspection can also consider broader quality attributes of a program*
 - ☐ such as compliance with standards, portability and maintainability.
 - ☐ You can look for inefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.
- There have been several studies and experiments that have demonstrated that inspections are more effective for defect discovery than program testing.
 - Fagan (Fagan, 1986) reported that more than 60% of the errors in a program can be detected using informal program inspections. Mills et al. (Mills, et al., 1987) suggest that a more formal approach to inspection based on correctness arguments can detect more than 90% of the errors in a program.
 - This technique is used in the Cleanroom process. Selby and Basili (Selby, et al., 1987) empirically compared the effectiveness of inspections and testing.
 - They found that static code reviewing was more effective and less expensive than defect testing in discovering program faults.
 - Gilb and Graham (Gilb and Graham, 1993) have also found this to be true.
 - Reviews and testing each have advantages and disadvantages and should be used together in the verification and validation process.
 - Gilb and Graham (Gilb and Graham, 1993) have also found this to be true that static code reviewing was more effective and less expensive.
 - They suggest that one of the most effective uses of reviews is to review the test cases for a system. Reviews can discover problems with these tests and can help design more effective ways to test the system.
 - You can start system V & V with inspections early in the development process, but once a system is integrated, you need testing to check its emergent properties and that the system's functionality is what the owner of the system really wants.

Roles in the inspection process with Description are as follow:

1. *Author or owner*

- ☐ The programmer or designer responsible for producing the program or document.

- ☐ Responsible for fixing defects discovered during the inspection process.

2. *Inspector*

- ☐ Finds errors, omissions and inconsistencies in programs and documents.
- ☐ May also identify broader issues that are outside the scope of the inspection team.

3. *Reader*

- ☐ Presents the code or document at an inspection meeting.

4. *Scribe*

- ☐ Records the results of the inspection meeting.

5. *Chairman or moderator*

- ☐ Manages the process and facilitates the inspection.
- ☐ Reports process results to the chief moderator.

6. *Chief moderator*

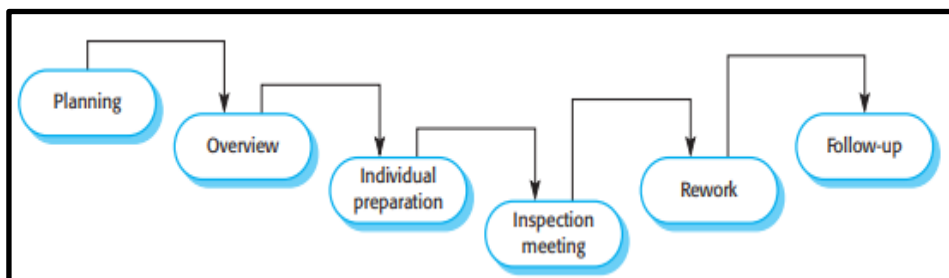
- ☐ Responsible for inspection process improvements, checklist updating, standards development, etc.

The activities in the inspection process are as follow:

Before a program inspection process begins, it is essential that,

1. You have a precise specification of the code to be inspected. It is impossible to inspect a component at the level of detail required to detect defects without a complete specification.
2. The inspection team members are familiar with the organisational standards.
3. An up-to-date, compilable version of the code has been distributed to all team members. There is no point in inspecting code that is ‘almost complete’ even if a delay causes schedule disruption.

The inspection process:



- During an inspection, a checklist of common programmer errors is often used to focus the discussion.

- You need different checklists for different programming languages because each language has its own characteristic errors.
- Humphrey (Humphrey, 1989), in a comprehensive discussion of inspections, gives a number of examples of inspection checklists.
- This checklist varies according to programming language because of the different levels of checking provided by the language compiler.
- For example, a Java compiler checks that functions have the correct number of parameters, a C compiler does not.
- Gilb and Graham (Gilb and Graham, 1993) emphasise that each organisation should develop its own inspection checklist based on local standards and practices.

Inspection checks with different Fault class are as follow:

1. *Data faults:*

- ☐ Are all program variables initialized before their values are used?
- ☐ Have all constants been named?
- ☐ Should the upper bound of arrays be equal to the size of the array or Size -1?
- ☐ If character strings are used, is a delimiter explicitly assigned?
- ☐ Is there any possibility of buffer overflow?

2. *Control faults:*

- ☐ For each conditional statement, is the condition correct?
- ☐ Is each loop certain to terminate?
- ☐ Are compound statements correctly bracketed?
- ☐ In case statements, are all possible cases accounted for?
- ☐ If a break is required after each case in case statements, has it been included?

3. *Input/output faults:*

- ☐ Are all input variables used?
- ☐ Are all output variables assigning a value before they are output?
- ☐ Can unexpected inputs cause corruption?
- ☐ *Interface faults:*
- ☐ Do all function and method calls have the correct number of parameters?
- ☐ Do formal and actual parameter types match?
- ☐ Are the parameters in the right order?

- ☐ If components access shared memory, do they have the same model of the shared memory structure?

4. *Storage management faults:*

- ☐ If a linked structure is modified, have all links been correctly reassigned?
- ☐ If dynamic storage is used, has space been allocated correctly?
- ☐ Is space explicitly de-allocated after it is no longer required?

5. *Exception management faults:*

- ☐ Have all possible error conditions been taken into account?

14.3 Automated Static Analysis

- Inspections are one form of static analysis where you examine the program without executing it.
- Inspections are often driven by checklists of errors and heuristics that identify common errors in different programming languages.
- For some errors and heuristics(an approach to problem solving or self-discovery), it is possible to automate the process of checking programs against this list, which has resulted in the development of automated static analyzers for different programming languages.
- Static analyzers are software tools that scan the source text of a program and detect possible faults and anomalies.
- They parse the program text and thus recognize the types of statements in the program.
- They can then detect whether statements are well formed, make inferences about the control flow in the program and, in many cases, compute the set of all possible values for program data.
- They complement the error detection facilities provided by the language compiler.
- They can be used as part of the inspection process or as a separate V & V process activity.
- The intention of automatic static analysis is to draw an inspector's attention to anomalies in the program, such as variables that are used without initialization, variables that are unused or data whose value could go out of range.

The stages involved in static analysis include:**1. Control flow analysis**

- ☐ This stage identifies and highlights loops with multiple exit or entry points and unreachable code.
- ☐ Unreachable code is code that is surrounded by unconditional go to statements or that is in a branch of a conditional statement where the guarding condition can never be true.

2. Data use analysis

- ☐ This stage highlights how variables in the program are used.
- ☐ It detects variables that are used without previous initialization, variables that are written twice without an intervening assignment and variables that are declared but never used.
- ☐ Data use analysis also discovers ineffective tests where the test condition is redundant. Redundant conditions are conditions that are either always true or always false.

3. Interface analysis

- ☐ This analysis checks the consistency of routine and procedure declarations and their use.
- ☐ It is unnecessary if a strongly typed language such as Java is used for implementation as the compiler carries out these checks.
- ☐ Interface analysis can detect type errors in weakly typed languages like FORTRAN and C.
- ☐ Interface analysis can also detect functions and procedures that are declared and never called or function results that are never used.

4. Information flow analysis

- ☐ This phase of the analysis identifies the dependencies between input and output variables.
- ☐ While it does not detect anomalies, it shows how the value of each program variable is derived from other variable values.
- ☐ With this information, a code inspection should be able to find values that have been wrongly computed.
- ☐ Information flow analysis can also show the conditions that affect a variable's value.

5. Path analysis

- ☐ This phase of semantic analysis identifies all possible paths through the program and sets out the statements executed in that path.
- ☐ It essentially unravels the program's control and allows each possible predicate to be analyzed individually.

Automated static analysis checks with Fault class are as follow:*1. Data faults*

- ☐ Variables used before initialization
- ☐ Variables declared but never used
- ☐ Variables assigned twice but never used between assignments
- ☐ Possible array bound violations
- ☐ Undeclared variables

2. Control faults

- ☐ Unreachable code
- ☐ Unconditional branches into loops

3. Input/output faults

- ☐ Variables output twice with no intervening assignment

4. Interface faults

- ☐ Parameter type mismatches
- ☐ Parameter number mismatches
- ☐ Non-usage of the results of functions
- ☐ Uncalled functions and procedures

5. Storage management faults

- ☐ Unassigned pointers
- ☐ Pointer arithmetic

14.4 Verification and Formal Methods

- Formal methods of software development are based on mathematical representations of the software, usually as a formal specification.
- These formal methods are mainly concerned with a mathematical analysis of the specification
- with transforming the specification to a more detailed, semantically equivalent representation; or
- with formally verifying that one representation of the system is semantically equivalent to another representation.
- We can use formal methods as the ultimate static verification technique.
- They require very detailed analyses of the system specification and the program, and their use is often time consuming and expensive.

- Consequently, the use of formal methods is mostly confined to safety- and security-critical software development processes.
- The use of formal mathematical specification and associated verification was mandated in UK defense standards for safety-critical software (MOD, 1995).

Formal methods may be used at different stages in the V & V process as shown below:

1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions.
 2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification and is effective in discovering programming and some design errors. A transformational development process where a formal specification is transformed through a series of more detailed representations or a Cleanroom process may be used to support the formal verification process.
- Formal verification demonstrates that the developed program meets its specification so implementation errors do not compromise dependability.
 - The argument against the use of formal specification is that it requires specialized notations.
 - These can only be used by specially trained staff and cannot be understood by domain experts.
 - Software engineers cannot recognize potential difficulties with the requirements because they don't understand the domain; domain experts cannot find these problems because they don't understand the specification.
 - Although the specification may be mathematically consistent, it may not specify the system properties that are really required.
 - Many people think that formal verification is not cost-effective.
 - The same level of confidence in the system can be achieved more cheaply by using other validation techniques such as inspections and system testing.
 - It is sometimes claimed that the use of formal methods for system development leads to more reliable and safer systems.
 - There is no doubt that a formal system specification is less likely to contain anomalies that must be resolved by the system designer.

Formal specification and proof do not guarantee that the software will be reliable in practical use. The reasons for this are:

1. *The specification may not reflect the real requirements of system users.*
 - ❑ Lutz (Lutz, 1993) discovered that many failures experienced by users were a consequence of specification errors and omissions that could not be detected by formal system specification.
 - ❑ System users rarely understand formal notations so they cannot read the formal specification directly to find errors and omissions.

2. *The proof may contain errors.*

- ❑ Program proofs are large and complex, so, like large and complex programs, they usually contain errors.

3. *The proof may assume a usage pattern which is incorrect.*

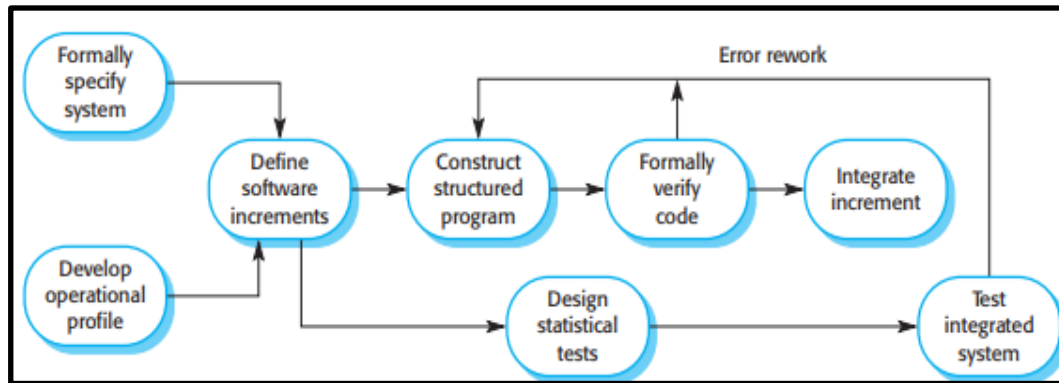
- ❑ If the system is not used as anticipated, the proof may be invalid.
- In spite of their disadvantages, formal methods have an important role to play in the development of critical software systems.
- Formal specifications are very effective in discovering specification problems that are the most common causes of system failure.
- Formal verification increases confidence in the most critical components of these systems.
- The use of formal approaches is increasing as procurers demand it and as more and more engineers become familiar with these techniques.

14.5 Cleanroom software development:

- Another well-documented approach that uses formal methods is the Cleanroom development process.
- Cleanroom software development (Mills, et al., 1987; Cobb and Mills, 1990; Linger, 1994; Prowell, et al., 1999) is a software development philosophy that uses formal methods to support rigorous software inspection.
- The **objective** of this approach to software development is **zero-defect software**.
- The name ‘Cleanroom’ was derived by analogy with semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.
- Cleanroom development is particularly relevant to this chapter because it has replaced the unit testing of system components by inspections to check the consistency of these components with their specifications.

The Cleanroom approach to software development is based on five key strategies:

The Cleanroom development process



1. Formal specification

- ☐ The software to be developed is formally specified.
- ☐ A state transition model that shows system responses to stimuli is used to express the specification.

2. Incremental development

- ☐ The software is partitioned into increments that are developed and validated separately using the Cleanroom process.
- ☐ These increments are specified, with customer input, at an early stage in the process.

3. Structured programming

- ☐ Only a limited number of control and data abstraction constructs are used.
- ☐ The program development process is a process of stepwise refinement of the specification.
- ☐ A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.

4. Static verification

- ☐ The developed software is statically verified using rigorous software inspections.
- ☐ There is no unit or module testing process for code components.

5. Statistical testing of the system

- ☐ The integrated software increment is tested statistically to determine its reliability.
- ☐ These statistical tests are based on an operational profile, which is developed in parallel with the system specification as shown in above Figure.

There are three teams involved when the Cleanroom process is used for large system development:

1. *The specification team*

- ☐ This group is responsible for developing and maintaining the system specification.
- ☐ This team produces customer-oriented specifications (the user requirements definition) and mathematical specifications for verification.
- ☐ In some cases, when the specification is complete, the specification team also takes responsibility for development.

2. *The development teams*

- ☐ This team has the responsibility of developing and verifying the software.
- ☐ The software is not executed during the development process.
- ☐ A structured, formal approach to verification based on inspection of code supplemented with correctness arguments is used.

3. *The certification teams*

- ☐ This team is responsible for developing a set of statistical tests to exercise the software after it has been developed.
- ☐ These tests are based on the formal specification.
- ☐ Test case development is carried out in parallel with software development.
- ☐ The test cases are used to certify the software reliability. Reliability growth models used to decide when to stop testing.

14.6 Summary:

- Verification and validation are not the same thing.
- Verification is intended to show that a program meets its specification.
- Validation is intended to show that the program does what the user requires.
- Static verification techniques involve examination and analysis of the program source code to detect errors.
- They should be used with program testing as part of the V & V process.
- Program inspections are effective in finding program errors.
- The aim of an inspection is to locate faults.
- A fault checklist should drive the inspection process.

- In a program inspection, a small team systematically checks the code.
- Team members include a team leader or moderator, the author of the code, a reader who presents the code during the inspection and a tester who considers the code from a testing perspective.
- Static analyzers are software tools that process a program source code and draw attention to anomalies such as unused code sections and uninitialized variables.
- These anomalies may be the result of faults in the code.
- Cleanroom software development relies on static techniques for program verification and statistical testing for system reliability certification.
- It has been successful in producing systems that have a high level of reliability.

14.7 Exercise

Answer the following:

1. Explain the differences between verification and validation, and explain why validation is a particularly difficult process.
2. Explain why program inspections are an effective technique for discovering errors in a program.
3. What types of errors are unlikely to be discovered through inspections?
4. Suggest why an organization with a competitive, elitist culture would probably find it difficult to introduce program inspections as a V & V technique.
5. Explain why it may be cost-effective to use formal methods in the development of safety critical software systems.
6. Why do you think that some developers of this type of system are against the use of formal methods?

SOFTWARE MEASUREMENT

Unit Structure

15.0 Objectives

15.1 Software Measurement - Introduction

15.1.1 Definition: Measurements

15.1.2 Software measurement and metrics

15.1.3 Software metric

15.1.4 Predictor and control measurements

15.1.5 Use of measurements

15.1.6 Metrics assumptions

15.1.7 Relationships between internal and external software

15.1.8 Problems with measurement in industry

15.1.9 Product metrics

15.1.9.1 Dynamic and static metrics

15.1.9.2 Static software product metrics

15.1.9.3 Static software product metrics

15.1.10 The CK object-oriented metrics suite

15.1.10.1 The CK object-oriented metrics suite

15.1.11 Software component analysis

15.1.12 The process of product measurement

15.1.13 Measurement surprises

15.2 Size Oriented Metrics

15.3 Function Oriented Metrics

15.3.1 Benefits of Function

15.3.2 How to Calculate Function

15.3.3 Counting function points

15.3.4 Functional units with weighting factors

15.3.5 The procedure for the calculation of Unadjusted Function Point (UFP)

15.4 Extended Function Point Metrics

15.5 Lets Sum up

15.6 References

15.7 Exercises

15.0 Objectives

This Chapter would make you understand the following concepts:

- What is Measurement?
- Software Measurement – Introduction
- Size Oriented Metrics
- Function Oriented Metrics
- Extended Function Point Metrics

15.1 Software Measurement - Introduction

15.1.1 Definition: Measurements in the **physical world** can be categorized in two ways:

- Direct measures and
Ex: length of a bolt
- Indirect measures
Ex: the “quality” of bolts produced, measured by counting rejects)

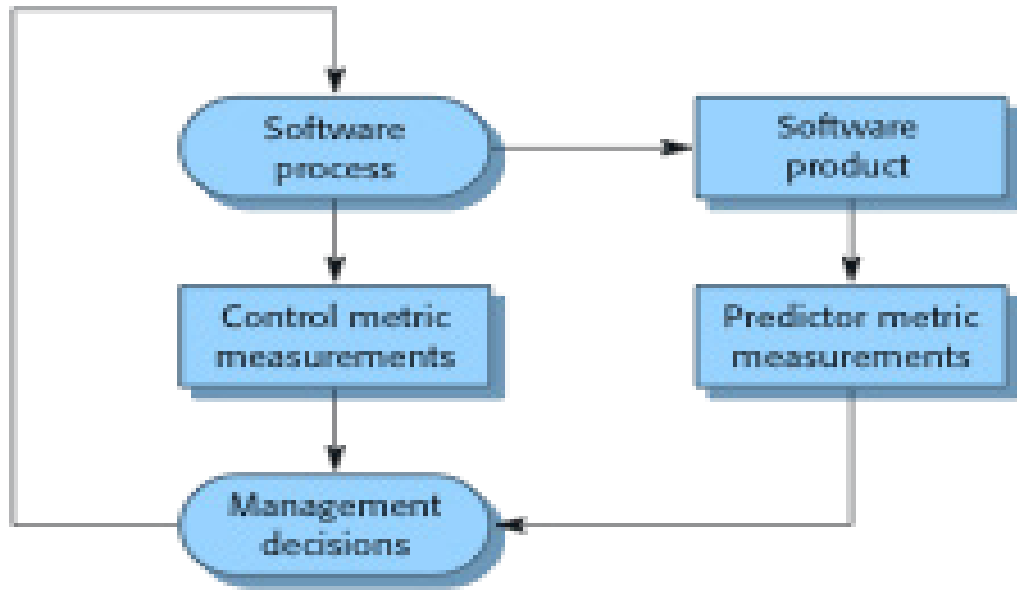
15.1.2 Software measurement and metrics

- **Definition:** Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process.
- Metrics is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- This allows for objective comparisons between techniques and processes.
- Although some companies have introduced measurement programmers, most organizations still don’t make systematic use of software measurement.
- There are few established standards in this area.

15.1.3 Software metric

- Any type of measurement which relates to a software system, process or related documentation
- Lines of code in a program, the Fog index, number of person days required to develop a component.
- Allow the software and the software process to be quantified.
- May be used to predict product attributes or to control the software process.
- Product metrics can be used for general predictions or to identify anomalous components.

15.1.4 Predictor and control measurements



15.1.5 Use of measurements

- To assign a value to system quality attributes
 - By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
- To identify the system components whose quality is substandard
 - Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

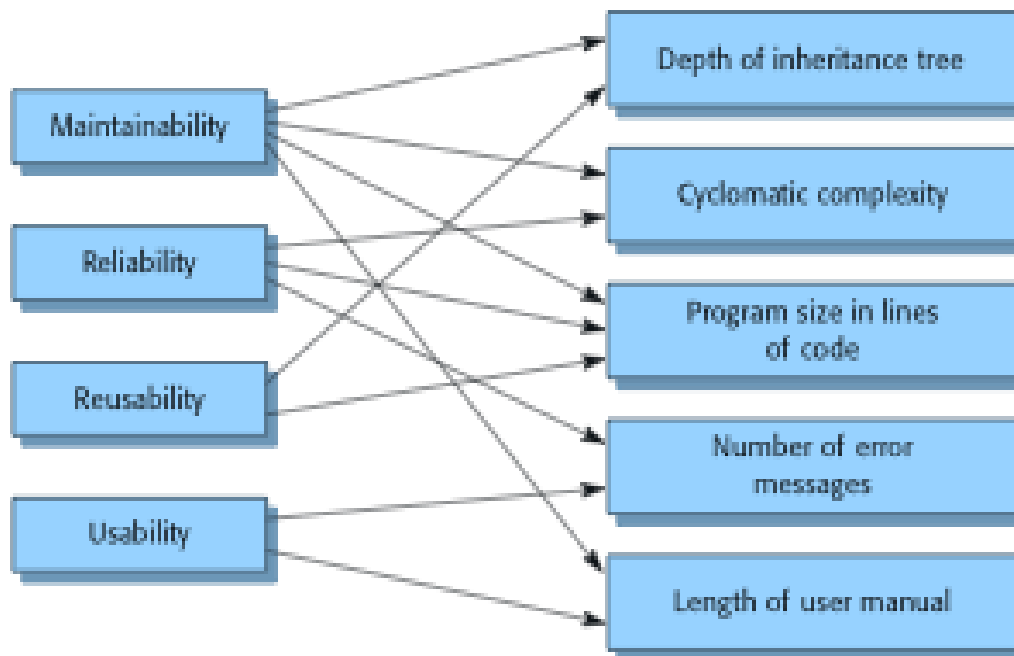
15.1.6 Metrics assumptions

- A software property can be measured.
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- This relationship has been formalized and validated.
- It may be difficult to relate what can be measured to desirable external quality attributes.

15.1.7 Relationships between internal and external software

External quality attributes

Internal attributes



15.1.8 Problems with measurement in industry

- It is impossible to quantify the return on investment of introducing an organizational metrics program.
- There are no standards for software metrics or standardized processes for measurement and analysis.
- In many companies, software processes are not standardized and are poorly defined and controlled.
- Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- Introducing measurement adds additional overhead to processes.

15.1.9 Product metrics

Definition: A quality metric should be a predictor of product quality.

Classes of product metric

- Dynamic metrics which are collected by measurements made of a program in execution;
- Static metrics which are collected by measurements made of the system representations;
- Dynamic metrics help assess efficiency and reliability
- Static metrics help assess complexity, understandability and maintainability.

15.1.9.1 Dynamic and static metrics

- Dynamic metrics are closely related to software quality attributes
 - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute).
- Static metrics have an indirect relationship with quality attributes
 - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability.

15.1.9.2 Static software product metrics

SOFTWARE METRIC	DESCRIPTION
FAN-IN/FAN-OUT	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
LENGTH OF CODE	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.

15.1.9.3 Static software product metrics

SOFTWARE METRIC	DESCRIPTION
CYCLOMATIC COMPLEXITY	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
LENGTH OF IDENTIFIERS	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
DEPTH OF CONDITIONAL NESTING	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.

FOG INDEX	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.
------------------	--

15.1.10 The CK object-oriented metrics suite

OBJECT-ORIENTED METRIC	DESCRIPTION
WEIGHTED METHODS PER CLASS (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as super classes in an inheritance tree.
DEPTH OF INHERITANCE TREE (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from super classes. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
NUMBER OF CHILDREN (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

15.1.10.1 The CK object-oriented metrics suite

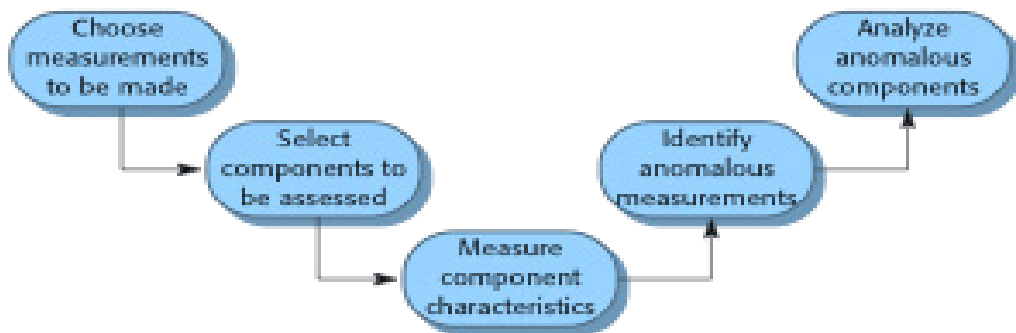
OBJECT-ORIENTED METRIC	DESCRIPTION
-------------------------------	--------------------

Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

15.1.11 Software component analysis

- System component can be analyzed separately using a range of metrics.
- The values of these metrics may then compared for different components and, perhaps, with historical measurement data collected on previous projects.
- Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

15.1.12 The process of product measurement



15.1.13 Measurement surprises

Reducing the number of faults in a program leads to an increased number of help desk calls

- The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase;
- A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls.

15.2 Size Oriented Metrics

Definition: Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced.

- If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure can be created. The table lists each software development project that has been completed over the past few years and corresponding measures for that project.

Referring to the table entry for project alpha:

Project	LOC	Effort	\$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

- 12, 100 lines of code were developed
- 24 person-months of effort
- at a cost of \$168, 000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding.
- Further information for project alpha indicates that
- 365 pages of documentation were developed,
- 134 errors were recorded before the software was released,

- 29 defects were encountered after release to the customer within the first year of operation.

Three people worked on the development of software for project alpha. In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value.

From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects 4 per KLOC.
- \$ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors person-month.
- LOC person-month.
- \$ per page of documentation.

15.3 Function Oriented Metrics

Definition: Functionalities, provided by the software is measured Independent of programming language used

15.3.1 Benefits of Function

- Function points are useful
- In measuring the size of the solution instead of the size of the problem.
- As requirements are the only thing needed for function points count.
- As it is independent of technology.

In **estimating testing**

- In estimating overall project costs, schedule and In contract negotiations as it provides a method of easier communication with business groups
- As it quantifies and assigns a value to the actual uses, interfaces, and purposes of the functions in the
- In creating ratios with other metrics such as hours, cost, headcount duration, and other application

15.3.2 How to Calculate Function

- Data for following – characteristics are collected.
- **Number of User Inputs** –
 - * each user input,

- * which provides, Distinct application data,
- * to the software
- * Is counted
- **Number of User Outputs -**
 - * each user output
 - * that provides – application data to the user – is counted.

Ex.: Screens, reports, error messages
- **Number of User Inquiries**
 - * an online input
 - * that results in
 - * the generation of some
 - * immediate software response
 - * in the form of an output
- **Number of Files**
 - * each logical master files
 - * **i.e.**, a logical grouping of data, that may be part of a database or a separate file
- **Number of External Interfaces**
 - * all machine-readable interfaces
 - * That are used to transmit information
 - * To another system
 - * are counted
- **The organization needs to develop criteria**
 - * Which determine
 - * Whether a particular entry is
 - * simple, average or complex
 - **The weighting factors**
 - * should be determined
 - * by observations or by experiments

The FPA functional units are shown in figure given below:

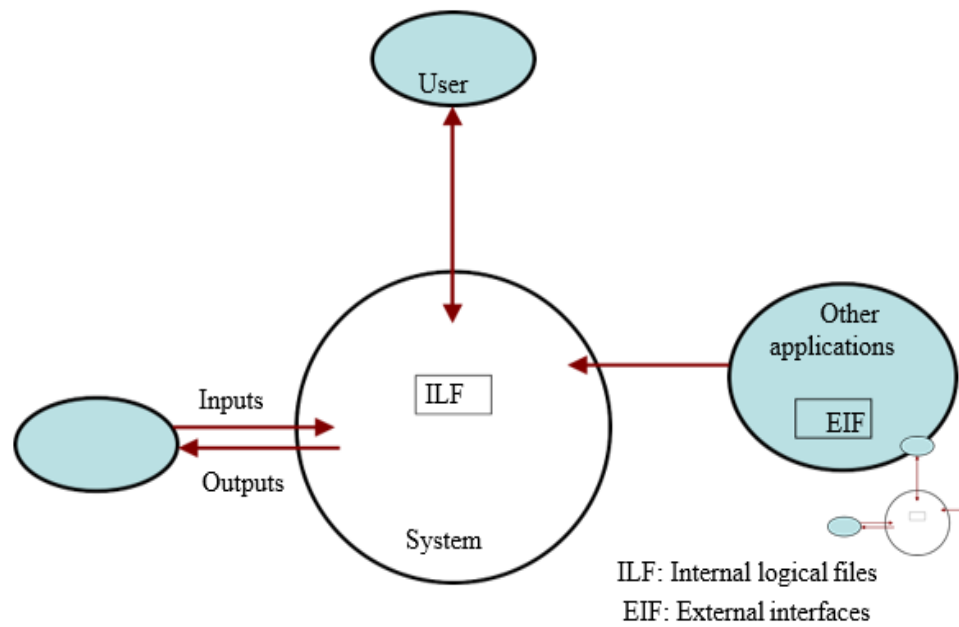


Fig. : FPAs functional units System

15.3.3 Counting function points

FUNCTIONAL UNITS	WEIGHTING FACTORS		
	LOW	AVERAGE	HIGH
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

15.3.4 Functional units with weighting factors

FUNCTIONAL UNITS	COUNT	COMPLEXITY	COMPLEXITY TOTALS	FUNCTIONAL UNIT TOTALS
External Inputs (EIs)	Low x 3	=		
	Average x 4	=		
	High x 6	=		

External Outputs (EOs)	Low x 4		=						
	Average x 5		=						
External Inquiries (EQs)		High x 7	=						
	Low x 3		=						
External Inquiries (EQs)	Average x 4		=						
		High x 6	=						
External logical Files (ILFs)	Low x 7		=						
	Average x 10		=						
	High x 15		=						
External Interface Files (EIFs)	Low x 5		=						
	Average x 7		=						
		High x 10	=						
Total Unadjusted Function Point Count									

Table: UFP calculation table

The weighting factors

- are identified for
- all functional units and
- Multiplied with the functional units accordingly.

15.3.5 The procedure for the calculation of Unadjusted Function Point (UFP) is given in table shown above.

Domain Characteristics	Count		Weighting Factor			Count
			low	avg	high	
Number Of User Inputs		*	3	4	6	

Number Of User Outputs		*	4	5	7	
Number Of User Enquiries		*	3	4	6	
Number Of Files		*	7	10	15	
Number Of External Interfaces		*	5	7	10	
Count Total						

- The Count Total – can be computed with the help of above given table
- Now – The Software Complexity –
 - -can be computed
 - -by answering following questions
- These are complexity adjustment values – (sum(Fi))

Table : Computing function points

Rate each factor on a scale of 0 to 5.

No Influence *Incidental* *Moderate* *Significant* *Essential*

Number of factors considered (F_i)

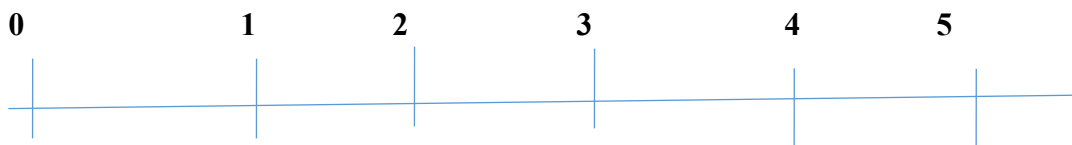
1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the online data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated online ?
9. Is the inputs, outputs, files, or inquiries complex ?

10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

- **Rate**

- * each of the above factors
- * according to the following scale of (0 to 5)

- **Function Points (FP) = count total * (0.65 + (0.01 * sum (Fi)))**



- **Rate each factor on a scale of 0 to 5.**

0 – No Influence

1 - Incidental

2 - Moderate

3 - Average

4 - Significant

5 – Essential

- Number of factors considered (F_i)
- These metrics are controversial and are not universally acceptable.
- There are standards issued by
- the International Functions Point User Group (IFPUG)
- and
- The United Kingdom Function Point User Group (UFPUG).
- An ISO standard for function point method is also being developed.

15.3.6Example: 1

- **Consider a project with the following functional units:**

Assume all complexity adjustment factors and weighting factors are average.

Compute the function points for the project.

Number of user inputs = 50

Number of user outputs = 40

Number of user enquiries = 35

Number of user files = 06

Number of external interfaces = 04

Domain Characteristics	Count		Weighting Factor			Count
			Simple	Average	Complex	
Number Of User Inputs	50	*		4		200
Number Of User Outputs	40	*		5		200
Number Of User Enquiries	35	*		4		140
Number Of Files	06	*		10		60
Number Of External Interfaces	04	*		7		28
Count Total			628			

Solution

We know

5 3

$$UFP = \sum_{i=1}^5 \sum_{j=1}^3 Z_{ij} w_{ij}$$

$$\begin{aligned} UFP &= (50 * 4) + (40 * 5) + (35 * 4) + (6 * 10) + (4 * 7) \\ &= 200 + 200 + 140 + 60 + 28 = 628 \end{aligned}$$

$$\begin{aligned} CAF &= (0.65 + 0.01 * \sum_i F_i) \\ &= (0.65 + 0.01 * (14 * 3)) \\ &= (0.65 + 0.01 * 42) \\ &= (0.65 + 0.42) \\ &= 1.07 \end{aligned}$$

$$FP = UFP * CAF = 628 * 1.07 = 672$$

Function Points (FP) =

count total * (0.65 + (0.01 * sum (Fi)))

$$FP = 628 * (0.65 + 0.01 * (14 * 3))$$

$$= 628 * (0.65 + 0.42)$$

$$= 628 * (1.07)$$

$$= 671.96$$

- Functions points may compute the following important metrics:
 - Avg productivity is → 6.5 FP per person-month
 - (i.e. 1 person works for 1 month – to develop 6.5 FP)
 - Avg labor cost is → Rs. 6000/- per month
1. Cost per function point = 6000/6.5 = Rs. 923 per function point
 2. Total estimated project cost = Rs. 923 * 672 FP = Rs. 6,20,256/-
 3. Total estimated effort = (672 / 6.5) = 103 person – month.
- i.e. 103 person will work for 1 month to complete the project.
- Or 52 person will work for 2 months to complete the project.
- Or 26 person will work for 4 months to complete the project.
- Or 13 person will work for 8 months to complete the project.

15.4 Extended Function Point Metrics

The function point measure was originally designed to be applied to business information

systems applications. To accommodate these applications, the data dimension (the information domain values discussed previously) was emphasized to the engineering, real-time, exclusion of the functional and behavioral (control) dimensions. For this reason, points are used for the end control-oriented applications.

Function point measure was inadequate for many engineering and embedded systems (which emphasize function and control). A number of extensions to the basic function point measure have been proposed to remedy this situation.

A function point extension called feature points [JON91], is a superset of the function point measure that can be applied to systems and engineering software applications.

The feature point measure accommodates applications in which algorithmic complexity is high. Real-time, process control and embedded software applications tend to have high algorithmic complexity and are therefore amenable to the feature point.

To compute the feature point, information domain values are again counted and weighted. The feature point metric counts a new software characteristic—algorithms. An algorithm is defined as "a bounded computational problem that is included within a specific computer program" [JON91]. Inverting a matrix, decoding a bit string, or handling an interrupt are all examples of algorithms.

A useful FAQ on function another function point extension for real-time systems and engineered products points (and extended has been developed by Boeing. The Boeing approach integrates the data dimension function points) can be obtained at of software with the functional and control dimensions to provide a function-oriented

[http://ourworld.](http://ourworld.compuserve.com/)

measure amenable to applications that emphasize function and control capabilities.

[compuserve.com/](http://ourworld.compuserve.com/)

Called the 3D function point [WHI95], characteristics of all three software dimensions

homepages/ softcomp/

are “counted, quantified, and transformed” into a measure that provides an indication of the functionality delivered by the software.

Counts of retained data (the internal program data structure; e.g., files) and external data (inputs, outputs, inquiries, and external references) are used along with measures of complexity to derive a data dimension count. The functional dimension is measured by considering “the number of internal operations required to transform input to output data” [WHI95]. For the purposes of 3D function point computation, a “transformation” is viewed as a series of processing steps that are constrained by a set of semantic statements. The control dimension is measured by counting the number of transitions between states.

A state represents some externally observable mode of behavior, and a transition occurs as a result of some event that causes the software or system to change its mode of behavior (i.e., to change state). For example, a wireless phone contains software that supports auto dial functions. To enter the auto-dial state from a resting state, the user presses an Auto key on the keypad. This event causes an LCD display to prompt for a code that will indicate the party to be called. Upon entry of the code and hitting the Dial key (another event), the wireless phone software makes a transition to the dialing state. When computing 3D function points, transitions are not assigned a complexity value.

To compute 3D function points, the following relationship is used:

$$\text{index} = \mathbf{I} + \mathbf{O} + \mathbf{Q} + \mathbf{F} + \mathbf{E} + \mathbf{T} + \mathbf{R} \quad (4-2)$$

It should be noted that other extensions to function points for application in real-time software work (e.g., [ALA97]) have also been proposed. However, none of these appears to be widely used in the industry.

Determining the statements complexity of a transformation for 3D function points [WHI95].

Processing steps

where **I, O, Q, F, E, T, and R** represent **complexity weighted values** for the elements discussed already: **inputs, outputs, inquiries, internal data structures, external files, transformation, and transitions**, respectively. Each complexity weighted value is computed using the following relationship:

where **N_{il}, N_{ia}, and N_{ih}** represent the number of occurrences of element **i** (e.g., out-puts) for each level of complexity (low, medium, high);

$$\text{Complexity weighted value} = \mathbf{N_{il} W_{il}} + \mathbf{N_{ia} W_{ia}} + \mathbf{N_{ih} W_{ih}}$$

SOFTWARE COST ESTIMATION

Unit Structure

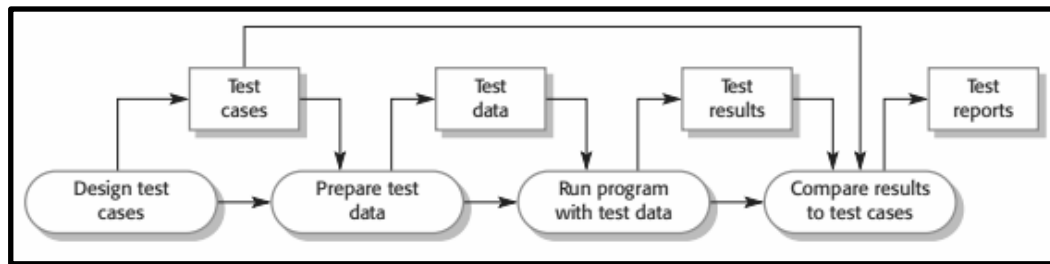
- 16.0 Objective
- 16.1 Software Testing:
- 16.2 System Testing
- 16.3 Component Testing
- 16.4 Test Case Design
- 16.5 Test Automation
- 16.6 Software Cost Estimation :
 - 16.6.1 Software Productivity
 - 16.6.2 Estimation Techniques
 - 16.6.3 Algorithmic Cost Modeling
 - 16.6.4 Project Duration and Staffing
- 16.7 Summary
- 16.8 Exercise

16.0 Objectives

- The Objective Of This Chapter Is To Describe The Processes Of Software Testing And Introduce A Range Of Testing Techniques.
- Understand The Distinctions Between Validation Testing and Defect Testing.
- Understand The Principles of System Testing and Component Testing.
- Understand Three Strategies That May Be Used to Generate System Test Cases.
- Understand The Essential Characteristics of Software Tools That Support Test Automation.
- The two fundamental testing activities are
 - component testing—testing the parts of the system and
 - system testing—testing the system as a whole.
- The Software Testing Process Has Two Distinct Goals:
 - To Demonstrate to The Developer and The Customer That the Software Meets Its Requirements
 - To Discover Faults or Defects In The Software Where The Behavior Of The Software Is Incorrect, Undesirable Or Does Not Conform To Its Specification.
- The Aim of The Component Testing Stage Is To Discover Defects By Testing Individual Program Components.

16.1 Software Testing:

A model of the software testing process



A general model of the testing process is shown in above Figure.

- Test cases are specifications of the inputs to the test and the expected output from the system plus a statement of what is being tested.
- Test data are the inputs that have been devised to test the system.
- Test data can sometimes be generated automatically. Automatic test case generation is impossible.
- The output of the tests can only be predicted by people who understand what the system should do.
- Exhaustive testing, where every possible program execution sequence is tested, is impossible.
- Testing, therefore, has to be based on a subset of possible test cases.
- Ideally, software companies should have policies for choosing this subset rather than leave this to the development team.
- These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once.
- Alternatively, the testing policies may be based on experience of system usage and may focus on testing the features of the operational system.

For example:

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g., text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

16.2 System Testing:

- System testing involves integrating two or more components that implement system functions or features and then testing this integrated system.

- In an iterative development process, system testing is concerned with testing an increment to be delivered to the customer; in a waterfall process, system testing is concerned with testing the entire system.

For most complex systems, there are two distinct phases to system testing:

1. *Integration testing*, where the test team has access to the source code of the system.

- When a problem is discovered, the integration team tries to find the source of the problem and identify the components that have to be debugged.
- Integration testing is mostly concerned with finding defects in the system.
- 2. *Release testing*, where a version of the system that could be released to users is tested.
- Here, the test team is concerned with validating that the system meets its requirements and with ensuring that the system is dependable.
- Release testing is usually ‘black-box’ testing where the test team is simply concerned with demonstrating that the system does or does not work properly.
- Problems are reported to the development team whose job is to debug the program.
- Where customers are involved in release testing, this is sometimes called acceptance testing. If the release is good enough, the customer may then accept it for use.

Integration testing

- The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions.
- The components that are integrated may be off-the-shelf components, reusable components that have been adapted for a particular system or newly developed components.
- Integration testing checks that these components actually work together, are called correctly and transfer the right data at the right time across their interfaces.
- System integration involves identifying clusters of components that deliver some system functionality and integrating these by adding code that makes them work together.
- In *the top-down integration* skeleton of the system is developed first, and components are added to it.
- Alternatively, in *the bottom-up integration* first integrate infrastructure components that provide common services, such as network and database

access, then add the functional components.

- In both top-down and bottom-up integration, usually we have to develop additional code to simulate other components and allow the system to execute.
- **Drawback:**
 - A major problem that arises during integration testing is localizing errors.
 - There are complex interactions between the system components and, when an anomalous output is discovered, you may find it hard to identify where the error occurred.
- To make it easier to locate errors, you should always use an incremental approach to system integration and testing. Initially, you should integrate a minimal system
- **Regression testing:**
 - These problems mean that when a new increment is integrated, it is important to rerun the tests for previous increments as well as the new tests that are required to verify the new system functionality.
 - Rerunning an existing set of tests is called regression testing.
 - If regression testing exposes problems, then you have to check whether these are problems in the previous increment that the new increment has exposed or whether these are due to the added increment of functionality.
 - Regression testing is clearly an expensive process and is impractical without some automated support.

Release testing:

- Release testing is the process of testing a release of the system that will be distributed to customers.
- The primary goal of this process is to increase the supplier's confidence that the system meets its requirements. If so, it can be released as a product or delivered to the customer.
- Release testing is usually a black-box testing process where the tests are derived from the system specification.

Performance testing:

- Once a system has been completely integrated, it is possible to test the system for emergent properties such as performance and reliability.
- Performance tests have to be designed to ensure that the system can process its intended load.
- This usually involves planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering

- problems and defects in the system.
- To test whether performance requirements are being achieved, you may have to construct an operational profile.

16.3 Component Testing:

- Component testing also known as unit testing is the process of testing individual components in the system.
- This is a defect testing process so its goal is to expose faults in these components.
- As I discussed in the introduction, for most systems, the developers of components are responsible for component testing.
- **There are different types of components that may be tested at this stage:**
 1. Individual functions or methods within an object
 2. Object classes that have several attributes and methods
 3. Composite components made up of several different objects or functions.
- These composite components have a defined interface that is used to access their functionality.
- Individual functions or methods are the simplest type of component and your tests are a set of calls to these routines with different input parameters.
- You can use the approaches to test case design, discussed in the next section, to design the function or method tests.
- When you are testing object classes, you should design your tests to provide coverage of all of the features of the object.
- Therefore, object class testing should include:
 1. The testing in isolation of all operations associated with the object
 2. The setting and interrogation of all attributes associated with the object
 3. The exercise of the object in all possible states.
- In principle, you should test every possible state transition sequence, although in practice this may be too expensive.
- Examples of state sequences that should be tested in the weather station include:
 - Shutdown → Waiting → Shutdown
 - Waiting → Calibrating → Testing → Transmitting → Waiting
 - Waiting → Collecting → Waiting → Summarizing → Transmitting → Waiting

16.4 Test Case Design:

- Test case design is a part of system and component testing where you design the test cases (inputs and predicted outputs) that test the system.
- The goal of the test case design process is to create a set of test cases that are effective in discovering program defects and showing that the system meets its requirements.
- To design a test case, you select a feature of the system or component that you are testing.
- You then select a set of inputs that execute that feature, document the expected outputs or output ranges and, where possible, design an automated check that tests that the actual and expected outputs are the same.
- There are various approaches that you can take to test case design:
 1. Requirements-based testing where test cases are designed to test the system requirements. This is mostly used at the system-testing stage as system requirements are usually implemented by several components. For each requirement, you identify test cases that can demonstrate that the system meets that requirement.
 2. Partition testing where you identify input and output partitions and design tests so that the system executes inputs from all partitions and generates outputs in all partitions. Partitions are groups of data that have common characteristics such as all negative numbers, all names less than 30 characters, all events arising from choosing items on a menu, and so on.
 3. Structural testing where you use knowledge of the program's structure to design tests that exercise all parts of the program. Essentially, when testing a program, you should try to execute each statement at least once. Structural testing helps identify test cases that can make this possible.
- In general, when designing test cases, you should start with the highest-level tests from the requirements then progressively add more detailed tests using partition and structural testing.

16.4.1 Requirements-based testing:

A general principle of requirements engineering is that requirements should be testable.

That is, the requirement should be written in such a way that a test can be designed so that an observer can check that the requirement has been satisfied.

Requirements-based testing, therefore, is a systematic approach to test case design

where you consider each requirement and derive a set of tests for it.

Requirements-based testing is validation rather than defect testing demonstrates that the system has properly implemented its requirements.

Process of Requirements based Testing:

- Defining Test Completion Criteria -
- Testing is completed only when all the functional and non-functional testing is complete.
- Design Test Cases -
- A Test case has five parameters namely the initial state or precondition, data setup, the inputs, expected outcomes and actual outcomes.
- Execute Tests -
- Execute the test cases against the system under test and document the results.
- Verify Test Results -
- Verify if the expected and actual results match each other.
- Verify Test Coverage -
- Verify if the tests cover both functional and non-functional aspects of the requirement.
- Track and Manage Defects -
- Any defects detected during the testing process goes through the defect life cycle and are tracked to resolution. Defect Statistics are maintained which will give us the overall status of the project.

Requirements Testing process:

- Testing must be carried out in a timely manner.
- Testing process should add value to the software life cycle, hence it needs to be effective.
- Testing the system exhaustively is impossible hence the testing process needs to be efficient as well.
- Testing must provide the overall status of the project; hence it should be manageable.
 - The Requirements based testing process starts at the very early phase of the software development, as correcting issues/errors is easier at this phase.
 - It begins at the requirements phase as the chances of occurrence of bugs have its roots here.
 - It aims at quality improvement of requirements. Insufficient requirements leads to failed projects.

16.4.2 Partition testing:

One systematic approach to test case design is based on identifying all partitions for a system or component.

Test cases are designed so that the inputs or outputs lie within these partitions.

Partition testing can be used to design test cases for both systems and components.

Input equivalence partitions are sets of data where all of the set members should be processed in an equivalent way.

Output equivalence partitions are program outputs that have common characteristics, so they can be considered as a distinct class.

You also identify partitions where the inputs are outside the other partitions that you have chosen.

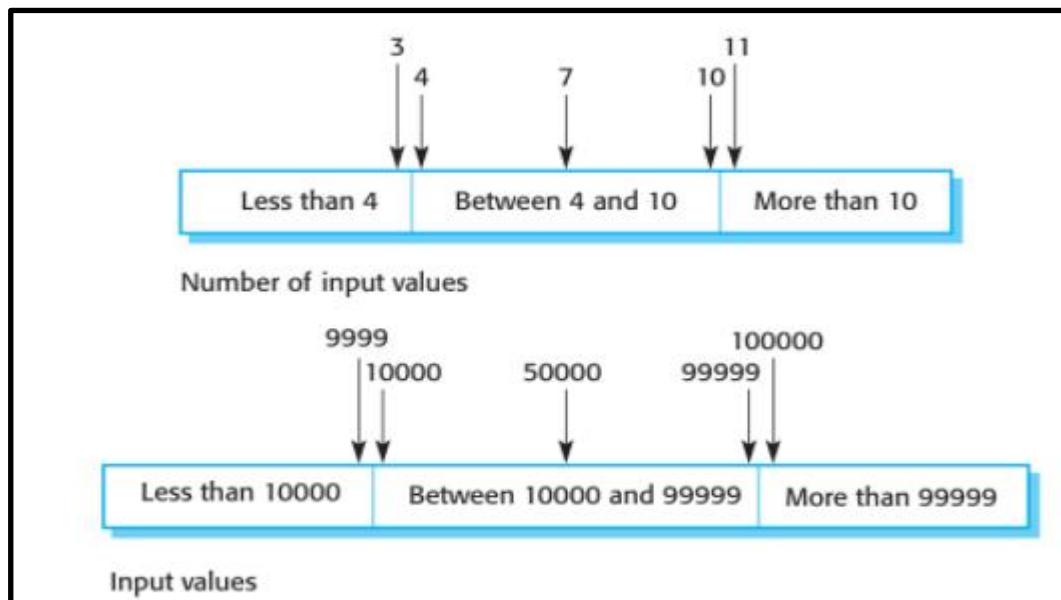
These test whether the program handles invalid input correctly.

Valid and invalid inputs also form equivalence partitions.

You identify partitions by using the program specification or user documentation and, from experience, where you predict the classes of input value that are likely to detect errors.

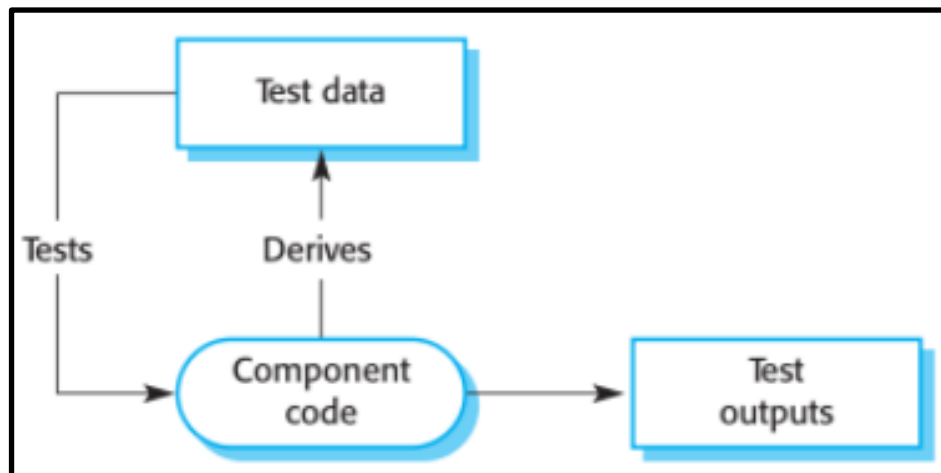
For example, say a program specification states that the program accepts 4 to 8 inputs that are five-digit integers greater than 10,000.

Equivalence partitions



16.4.3 Structural testing:

- Structural testing is an approach to test case design where the tests are derived from knowledge of the software's structure and implementation.
- This approach is sometimes called 'white-box', 'glass-box' testing, or 'clear-box' test-
- ing to distinguish it from black-box testing.
- Understanding the algorithm used in a component can help you identify further partitions and test cases.



-
- Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.
- Structural Testing Techniques:
 - Statement Coverage - This technique is aimed at exercising all programming statements with minimal tests.
 - Branch Coverage - This technique is running a series of tests to ensure that all branches are tested at least once.
 - Path Coverage - This technique corresponds to testing all possible paths which means that each statement and branch are covered.

16.4.4 Path testing:

- Path Testing is a structural testing method based on the source code or algorithm and NOT based on the specifications.
- It can be applied at different levels of granularity.
- Path Testing Assumptions:
 - The Specifications are Accurate
 - The Data is defined and accessed properly
 - There are no defects that exist in the system other than those that affect control flow
- Path Testing Techniques:
 - Control Flow Graph (CFG) - The Program is converted into Flow graphs by representing the code into nodes, regions and edges.
 - Decision to Decision path (D-D) - The CFG can be broken into various Decision to Decision paths and then collapsed into individual nodes.
 - Independent (basis) paths - Independent path is a path through a DD-path graph which cannot be reproduced from other paths by other methods.

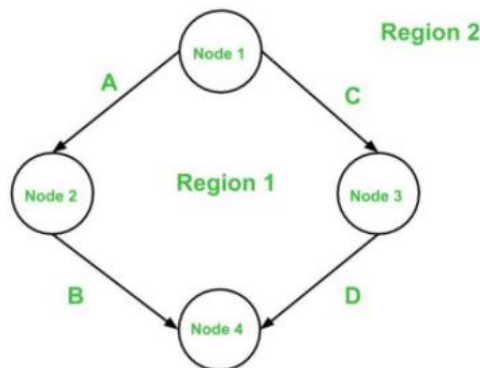
1. Control Flow Graph –

A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module. A control flow graph (V, E) has V number of nodes/vertices and E number of edges in it. A control graph can also have :

Junction Node – a node with more than one arrow entering it.

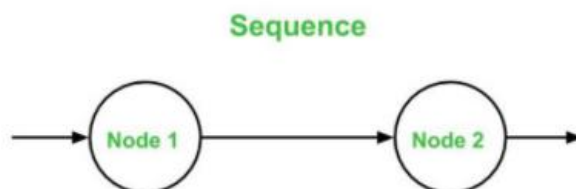
Decision Node – a node with more than one arrow leaving it.

Region – area bounded by edges and nodes (area outside the graph is also counted as a region.).



Below are the **notations** used while constructing a flow graph :

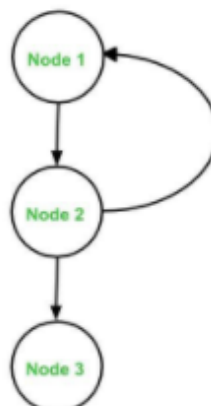
- **Sequential Statements –**



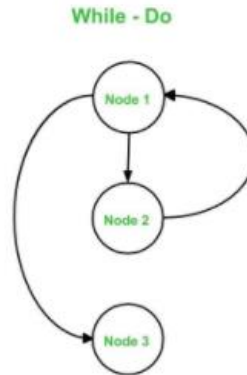
- **If – Then – Else –**

- **Do – While –**

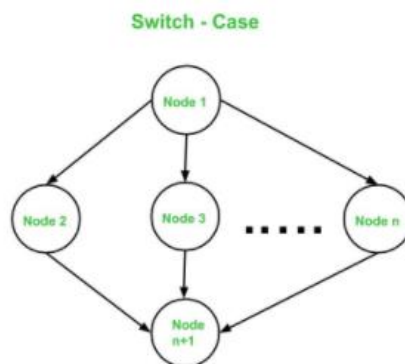
Do - While



- **While – Do –**



- **Switch – Case –**

**Cyclomatic Complexity –**

The cyclomatic complexity $V(G)$ is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae :

Formula based on edges and nodes :

$$V(G) = e - n + 2 * P$$

Where,

e is number of edges,

n is number of vertices,

P is number of connected components.

For example, consider first graph given above,

where, $e = 4$, $n = 4$ and $p = 1$

So,

Cyclomatic complexity $V(G)$

$$= 4 - 4 + 2 * 1$$

$$1. \quad = 2$$

Formula based on Decision Nodes :

$$V(G) = d + P$$

where,

d is number of decision nodes,

P is number of connected nodes.

For example, consider first graph given above,

where, $d = 1$ and $p = 1$

So,

Cyclomatic Complexity $V(G)$

$$= 1 + 1$$

$$2. \quad = 2$$

Formula based on regions :

$V(G)$ = number of regions in the graph

For example, consider first graph given above,

Cyclomatic complexity $V(G)$

$$= 1 \text{ (for region 1)} + 1 \text{ (for Region 2)}$$

$$3. \quad = 2$$

Hence, using all the three above formulae, the cyclomatic complexity obtained remains same. All these three formulae can be used to compute and verify the cyclomatic complexity of the flow graph.

Note –

1. For one function [e.g., Main() or Factorial()], only one flow graph is constructed. If in a program, there are multiple functions, then a separate flow graph is constructed for each one of them. Also, in the cyclomatic complexity formula, the value of ‘p’ is set depending of the number of graphs present in total.
2. If a decision node has exactly two arrows leaving it, then it is counted as one decision node. However, if there are more than 2 arrows leaving a decision node, it is computed using this formula :

$$d = k - 1$$

Here, k is number of arrows leaving the decision node.

Independent Paths :

An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined. The cyclomatic complexity gives the number of independent paths present in a flow graph. This is because the cyclomatic complexity is used as an upper-bound for the number of tests that should be executed in order to make sure that all the statements in the program have been executed at least once.

Consider first graph given above here the independent paths would be 2 because number of independent paths is equal to the cyclomatic complexity.

So, the independent paths in above first given graph :

- Path 1:

A -> B

- Path 2:

C -> D

Note –

Independent paths are not unique. In other words, if for a graph the cyclomatic complexity comes out be N, then there is a possibility of obtaining two different sets of paths which are independent in nature.

Design Test Cases :

Finally, after obtaining the independent paths, test cases can be designed where each test case represents one or more independent paths.

Advantages :

Basis Path Testing can be applicable in the following cases:

1. More Coverage –

Basis path testing provides the best code coverage as it aims to achieve maximum logic coverage instead of maximum path coverage. This results in an overall thorough testing of the code.

2. Maintenance Testing –

When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing.

3. Unit Testing –

When a developer writes the code, he or she tests the structure of the program or module themselves first. This is why basis path testing requires enough knowledge about the structure of the code.

4. Integration Testing –

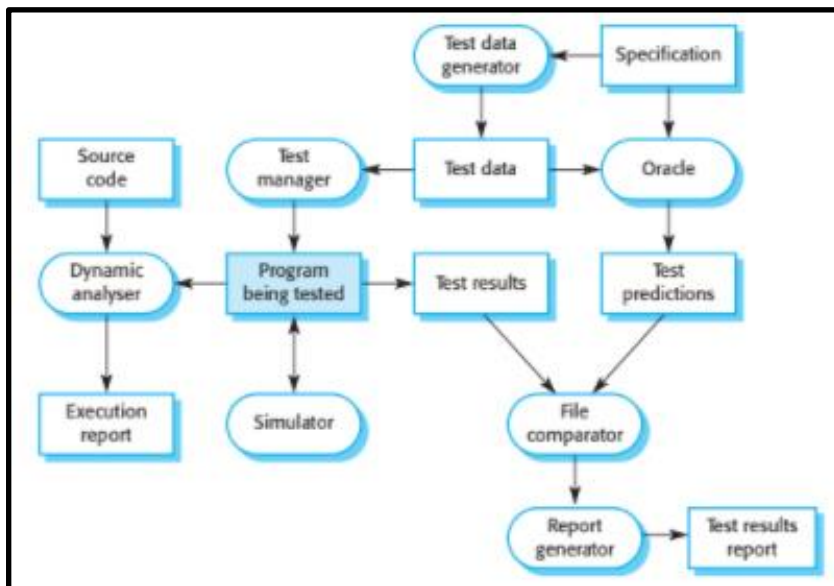
When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.

5. Testing Effort –

Since the basis path testing technique takes into account the complexity of the software (i.e., program or module) while computing the cyclomatic complexity, therefore it is intuitive to note that testing effort in case of basis path testing is directly proportional to the complexity of the software or program.

16.5 Test Automation:

- Testing is an expensive and laborious phase of the software process.
- As a result, testing tools were among the first software tools to be developed.
- These tools offer a range of facilities and their use can significantly reduce the costs of testing.
- The tests themselves should be written in such a way that they indicate whether the tested system has behaved as expected.
- A software testing workbench is an integrated set of tools to support the testing process.
- In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.
- Following Figure shows some of the tools that might be included in such a testing workbench:



1. **Test manager** Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.
2. **Test data generator** Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.
3. **Oracle** Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-to-back testing involves running the oracle and the program to be tested in parallel. Differences in their outputs are highlighted.

4. **File comparator** Compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.
5. **Report generator** Provides report definition and generation facilities for test results.
6. **Dynamic analyzer** Adds code to a program to count the number of times each statement has been executed. After testing, an execution profile is generated showing how often each program statement has been executed.
7. **Simulator** Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute.

User interface simulators are script-driven programs that simulate multiple simultaneous user

interactions. Using simulators for I/O means that the timing of transaction sequences is repeatable.

16.6 Software Cost Estimation:

It is always necessary to know how much any new project will cost to develop and how much development time will it take.

These estimates are needed before development is initiated.

The objective of this section is to introduce techniques for estimating the cost and effort required for software production.

- understand the fundamentals of software costing and reasons why the price of the software may not be directly related to its development cost;
- have been introduced to three metrics that are used for software productivity assessment;
- appreciate why a range of techniques should be used when estimating software costs and schedule;
- understand the principles of the COCOMO II model for algorithmic cost estimation.

Several estimation procedures have been developed and are having the following attributes in common.

1. Project scope must be established in advanced.
2. Software metrics are used as a support from which evaluation is made.
3. The project is broken into small PCs which are estimated individually.

To achieve true cost & schedule estimate, several options arise.

4. Delay estimation
5. Used symbol decomposition techniques to generate project cost and schedule estimates.
6. Acquire one or more automated estimation tools.

16.6.1 Software productivity:

Factors affecting software pricing are as follow:

- **Market opportunity**
 - A development organization may quote a low price because it wishes to move into a new segment of the software market.
 - Accepting a low profit on one project may give the organization the opportunity to make a greater profit later.
 - The experience gained may also help it develop new products.
- **Cost estimate uncertainty**
 - If an organization is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
- **Contractual terms**
 - A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects.
 - The price charged may then be less than if the software source code is handed over to the customer.
- **Requirement volatility**
 - If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
- **Financial health**
 - Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.
- Productivity estimates are usually based on measuring attributes of the software and dividing this by the total effort required for development.
- There are two types of metrics that have been used:
-

1. *Size-related metrics*

- These are related to the size of some output from an activity.
- The most commonly used size-related metric is lines of delivered source code.
- Other metrics that may be used are the number of delivered object code instructions or the number of pages of system documentation.

2. *Function-related metrics*

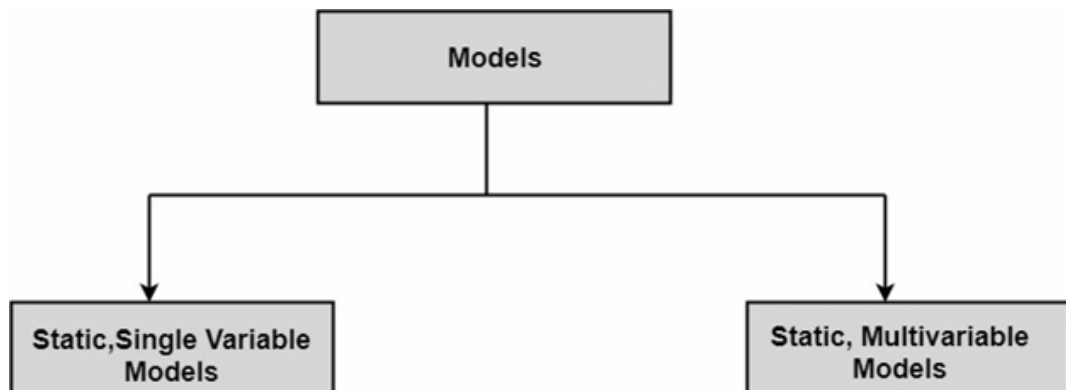
- These are related to the overall functionality of the delivered software. Productivity is expressed in terms of the amount of useful functionality produced in some given time.
- Function points and object points are the best-known metrics of this type.

Cost Estimation Models:

A model may be static or dynamic.

In a static model, a single variable is taken as a key element for calculating cost and time.

In a dynamic model, all variable are interdependent, and there is no basic variable.



Static, Single Variable Models:

When a model makes use of single variables to calculate desired values such as cost, time, efforts, etc. is said to be a single variable model. The most common equation is:

$$C=aL^b$$

Where C = Costs

L= size

a and b are constants

The Software Engineering Laboratory established a model called SEL model, for estimating its software production. This model is an example of the static, single

variable model.

$$E=1.4L^{0.93}$$

$$DOC=30.4L^{0.90}$$

$$D=4.6L^{0.26}$$

Where E= Efforts (Person Per Month)

DOC=Documentation (Number of Pages)

D = Duration (D, in months)

L = Number of Lines per code

Static, Multivariable Models:

These models are based on method (1), they depend on several variables describing various aspects of the software development environment.

In some models, several variables are needed to describe the software development process, and the selected equation combines these variables to give the estimate of time & cost.

These models are called multivariable models.

WALSTON and FELIX develop the models at IBM provide the following equation gives a relationship between lines of source code and effort:

$$E=5.2L^{0.91}$$

In the same manner duration of development is given by

$$D=4.1L^{0.36}$$

The productivity index uses 29 variables which are found to be highly correlated productivity as follows:

$$I = \sum_{i=1}^{29} W_i X_i$$

Where W_i is the weight factor for the i^{th} variable and $X_i=\{-1,0,+1\}$ the estimator gives X_i one of the values **-1, 0 or +1** depending on the variable decreases, has no effect or increases the productivity.

Example: Compare the Walston-Felix Model with the SEL model on a software development expected to involve 8 person-years of effort.

- Calculate the number of lines of source code that can be produced.
- Calculate the duration of the development.
- Calculate the productivity in LOC/PY
- Calculate the average manning

Solution:

The amount of manpower involved = 8PY=96persons-months

(a)Number of lines of source code can be obtained by reversing equation to give:

$$L = \left(\frac{E}{a}\right)^{1/b}$$

Then

$$L(\text{SEL}) = (96/1.4)^{1/0.93} = 94264 \text{ LOC}$$

$$L(\text{SEL}) = (96/5.2)^{1/0.91} = 24632 \text{ LOC}$$

(b)Duration in months can be calculated by means of equation

$$\begin{aligned} D(\text{SEL}) &= 4.6 (L)^{0.26} \\ &= 4.6 (94.264)^{0.26} = 15 \text{ months} \end{aligned}$$

$$\begin{aligned} D(\text{W-F}) &= 4.1 L^{0.36} \\ &= 4.1 (24.632)^{0.36} = 13 \text{ months} \end{aligned}$$

(c) Productivity is the lines of code produced per persons/month (year)

$$P(\text{SEL}) = \frac{94264}{8} = 11783 \frac{\text{LOC}}{\text{Person}} - \text{Years}$$

$$P(\text{Years}) = \frac{24632}{8} = 3079 \frac{\text{LOC}}{\text{Person}} - \text{Years}$$

(d)Average manning is the average number of persons required per month in the project

$$M(\text{SEL}) = \frac{96P-M}{15M} = 6.4 \text{ Persons}$$

$$M(\text{W-F}) = \frac{96P-M}{13M} = 7.4 \text{ Persons}$$

COCOMO Model :

- Boehm proposed COCOMO (Constructive Cost Estimation Model) in 1981.
- COCOMO is one of the most generally used software estimation models in the world. COCOMO predicts the efforts and schedule of a software product based on the size of the software.
- **The necessary steps in this model are:**
 - Get an initial estimate of the development effort from evaluation of thousands of delivered lines of source code (KDLOC).
 - Determine a set of 15 multiplying factors from various attributes of the project.
 - Calculate the effort estimate by multiplying the initial estimate with all the multiplying factors i.e., multiply the values in step1 and step2.

- The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single variable models, using KDLOC as the measure of the size.
- To determine the initial effort E_i in person-months the equation used is of the type is shown below
- $E_i = a * (KDLOC)^b$, where the value of the constant a and b are depends on the project type.

In COCOMO, projects are categorized into three types:

1. Organic
2. Semidetached
3. Embedded

1.Organic:

- A development project can be treated of the organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar methods of projects.
- Examples of this type of projects are simple business systems, simple inventory management systems, and data processing systems.

2. Semidetached:

- A development project can be treated with semidetached type if the development consists of a mixture of experienced and inexperienced staff.
- Team members may have finite experience in related systems but may be unfamiliar with some aspects of the order being developed.
- Example of Semidetached system includes developing a new operating system (OS), a Database Management System (DBMS), and complex inventory management system.

3. Embedded:

- A development project is treated to be of an embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational method exist.
- For Example: ATM, Air Traffic control.
- For three product categories, Bohem provides a different set of expression to predict effort (in a unit of person month)and development time from the size of estimation in KLOC(Kilo Line of code) efforts estimation takes into account the productivity loss due to holidays, weekly off, coffee breaks, etc.

According to Boehm, software cost estimation should be done through three stages:

1. Basic Model
2. Intermediate Model
3. Detailed Model

1. Basic COCOMO Model:

The basic COCOMO model provide an accurate size of the project parameters.

The following expressions give the basic COCOMO estimation model:

$$\text{Effort} = a_1 * (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 * (\text{efforts})^{b_2} \text{ Months}$$

Where

KLOC is the estimated size of the software product indicate in Kilo Lines of Code,

a_1, a_2, b_1, b_2 are constants for each group of software products,

Tdev is the estimated time to develop the software, expressed in months,

Effort is the total effort required to develop the software product, expressed in person months (PMs).

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic: $\text{Effort} = 2.4(\text{KLOC})^{1.05} \text{ PM}$

Semi-detached: $\text{Effort} = 3.0(\text{KLOC})^{1.12} \text{ PM}$

Embedded: $\text{Effort} = 3.6(\text{KLOC})^{1.20} \text{ PM}$

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic: $\text{Tdev} = 2.5(\text{Effort})^{0.38} \text{ Months}$

Semi-detached: $\text{Tdev} = 2.5(\text{Effort})^{0.35} \text{ Months}$

Embedded: $\text{Tdev} = 2.5(\text{Effort})^{0.32} \text{ Months}$

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes.

Following Fig shows a plot of estimated effort versus product size.

From fig, we can observe that the effort is somewhat superliner in the size of the software product.

Thus, the effort required to develop a product increases very rapidly with project size.

Example1: Suppose a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three model i.e., organic, semi-detached & embedded.

Solution: The basic COCOMO equation takes the form:

$$\text{Effort} = a_1 * (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 * (\text{efforts})^{b_2} \text{ Months}$$

$$\text{Estimated Size of project} = 400 \text{ KLOC}$$

(i) Organic Mode

$$E = 2.4 * (400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5 * (1295.31)^{0.38} = 38.07 \text{ PM}$$

(ii) Semidetached Mode

$$E = 3.0 * (400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5 * (2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded Mode

$$E = 3.6 * (400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5 * (4772.8)^{0.32} = 38 \text{ PM}$$

Example2: A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the Effort, development time, average staff size, and productivity of the project.

Solution: The semidetached mode is the most appropriate mode, keeping in view the size, schedule and experience of development time.

$$\text{Hence } E = 3.0(200)^{1.12} = 1133.12 \text{ PM}$$

$$D = 2.5(1133.12)^{0.35} = 29.3 \text{ PM}$$

$$\begin{aligned} \text{Average Staff Size (SS)} &= \frac{E}{D} \text{ Persons} \\ &= \frac{1133.12}{29.3} = 38.67 \text{ Persons} \end{aligned}$$

$$\text{Productivity} = \frac{\text{KLOC}}{E} = \frac{200}{1133.12} = 0.1765 \text{ KLOC/PM}$$

$$P = 176 \text{ LOC/PM}$$

2. Intermediate Model:

The basic Cocomo model considers that the effort is only a function of the number of lines of code and some constants calculated according to the various software systems.

The intermediate COCOMO model recognizes these facts and refines the initial estimates obtained through the basic COCOMO model by using a set of 15 cost drivers based on various attributes of software engineering.

Classification of Cost Drivers and their attributes:

(i) Product attributes -

- Required software reliability extent
- Size of the application database
- The complexity of the product

Hardware attributes -

- Run-time performance constraints
- Memory constraints
- The volatility of the virtual machine environment
- Required turnabout time

Personnel attributes -

- Analyst capability
- Software engineering capability
- Applications experience
- Virtual machine experience
- Programming language experience

Project attributes -

- Use of software tools
- Application of software engineering methods
- Required development schedule

Intermediate COCOMO equation:

$$E = a_i (\text{KLOC})^{b_i} \cdot EAF$$

$$D = c_i (E)^{d_i}$$

3. Detailed COCOMO Model:

Detailed COCOMO incorporates all qualities of the standard version with an assessment of the cost drivers' effect on each method of the software engineering process.

The detailed model uses various effort multipliers for each cost driver property.

In detailed cocomo, the whole software is differentiated into multiple modules, and

then we apply COCOMO in various modules to estimate effort and then sum the effort.

The Six phases of detailed COCOMO are:

1. Planning and requirements
2. System structure
3. Complete structure
4. Module code and test
5. Integration and test
6. Cost Constructive model

The effort is determined as a function of program estimate, and a set of cost drivers are given according to every phase of the software lifecycle.

16.6.2 Estimation techniques:

Organizations need to make software effort and cost estimates.

To do so, one or more of the techniques described below may be used.

All of these techniques rely on experience-based judgements by project managers who use their knowledge of previous projects to arrive at an estimate of the resources required for the project.

Some examples of the changes that may affect estimates based on experience include:

1. Distributed object systems rather than mainframe-based systems
2. Use of web services
3. Use of ERP or database-centered systems
4. Use of off-the-shelf software rather than original system development
5. Development for and with reuse rather than new development of all parts of a system
6. Development using scripting languages such as TCL or Perl (Ousterhout, 1998)
7. The use of CASE tools and program generators rather than unsupported software development.

Various Cost estimation techniques are as follows:

- Algorithmic cost modelling
 - A model is developed using historical cost information that relates some software metric (usually its size) to the project cost.
 - An estimate is made of that metric and the model predicts the effort required.

- **Expert judgement**
 - Several experts on the proposed software development techniques and the application domain are consulted.
 - They each estimate the project cost.
 - These estimates are compared and discussed.
 - The estimation process iterates until an agreed estimate is reached.
- **Estimation by analogy**
 - This technique is applicable when other projects in the same application domain have been completed.
 - The cost of a new project is estimated by analogy with these completed projects.
 - Myers (Myers, 1989) gives a very clear description of this approach.
- **Parkinson's Law**
 - Parkinson's Law states that work expands to fill the time available.
 - The cost is determined by available resources rather than by objective assessment.
 - If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
- **Pricing to win**
 - The software cost is estimated to be whatever the customer has available to spend on the project.
 - The estimated effort depends on the customer's budget and not on the software functionality.

16.6.3 Algorithmic cost modelling:

- Algorithmic cost modelling uses a mathematical expression to predict project costs based on estimates of the project size, the number of software engineers, and other process and product factors.
- An algorithmic cost model can be developed by analyzing the costs and attributes of completed projects and finding the closest fit mathematical expression to the actual project.
- In general, an algorithmic cost estimate for software cost can be expressed as:
- $EFFORT = A \times SIZE^B \times M$
- In this equation A is a constant factor that depends on local organizational practices and the type of software that is developed.
- Variable SIZE may be either the code size or the functionality of software expressed in function or object points.

- M is a multiplier made by combining process, product and development attributes, such as the dependability requirements for the software and the experience of the development team.
- The exponential component B associated with the size estimate expresses the non-linearity of costs with project size.
- As the size of the software increases, extra costs are emerged.
- The value of exponent B usually lies between 1 and 1.5.
- All algorithmic models have the same difficulties:
 - It is difficult to estimate SIZE in the early stage of development. Function or object point estimates can be produced easier than estimates of code size but are often inaccurate.
 - The estimates of the factors contributing to B and M are subjective. Estimates vary from one person to another person, depending on their background and experience with the type of system that is being developed.
- The number of lines of source code in software is the basic software metric used in many algorithmic cost models.
- The code size can be estimated by previous projects, by converting function or object points to code size, by using a reference component to estimate the component size, etc.
- The programming language used for system development also affects the number of lines of code to be implemented.
- Furthermore, it may be possible to reuse codes from previous projects and the size estimate has to be adjusted to take this into account.

16.6.4 Project Duration and Staffing:

- As well as estimating the effort required to develop a software system and the overall project costs, project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project.
- The development time for the project is called the project schedule.
- Increasingly, organizations are demanding shorter development schedules so that their products can be brought to market before their competitor's.
- The relationship between the number of staff working on a project, the total effort required and the development time is not linear.
- As the number of staff increases, more effort may be needed. The reason for this is that people spend more time communicating and defining interfaces between the parts of the system developed by other people.
- Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved.
- The COCOMO model includes a formula to estimate the calendar time (TDEV) required to complete a project. The time computation formula is the

same for all COCOMO levels:

- $TDEV = 3 (PM)(0.33+0.2*(B-1.01))$
- PM is the effort computation and B is the exponent computed, as discussed above (B is 1 for the early prototyping model).
- This computation predicts the nominal schedule for the project.
- However, the predicted project schedule and the schedule required by the project plan are not necessarily the same thing.
- The planned schedule may be shorter or longer than the nominal predicted schedule.
- However, there is obviously a limit to the extent of schedule changes, and the COCOMO II model predicts this:
- $TDEV = 3 (PM)(0.33+0.2*(B-1.01)) \text{ SCEDPercentage}/100$
- SCEDPercentage is the percentage increase or decrease in the nominal schedule.
- If the predicted figure then differs significantly from the planned schedule, it suggests that there is a high risk of problems delivering the software as planned.
- To illustrate the COCOMO development schedule computation, assume that 60 months of effort are estimated to develop a software system (Option C in Figure Assume that the value of exponent B is 1.17. From the schedule equation, the time required to complete the project is:
- $TDEV = 3 (60)0.36 = 13 \text{ months}$
- In this case, there is no schedule compression or expansion, so the last term in the formula has no effect on the computation.
- An interesting implication of the COCOMO model is that the time required to complete the project is a function of the total effort required for the project.
- It does not depend on the number of software engineers working on the project.
- This confirms the notion that adding more people to a project that is behind schedule is unlikely to help that schedule to be regained. Myers (Myers, 1989) discusses the problems of schedule acceleration.
- He suggests that projects are likely to run into significant problems if they try to develop software without allowing sufficient calendar time.

16.7 Summary:

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- Component testing is the responsibility of the component developer. A separate testing team usually carries out system testing.
- Integration testing is the initial system testing activity where you test integrated components for defects. Release testing is concerned with testing

customer releases and should validate that the system to be released meets its requirements.

- When testing systems, you should try to ‘break’ the system by using experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems.
- Interface testing is intended to discover defects in the interfaces of composite components. Interface defects may arise because of errors made in reading the specification, specification misunderstandings or errors or invalid timing assumptions.
- Equivalence partitioning is a way of deriving test cases. It depends on finding partitions in the input and output data sets and exercising the program with values from these
- partitions. Often, the value that is most likely to lead to a successful test is a value at the boundary of a partition.
- Structural testing relies on analyzing a program to determine paths through it and using this analysis to assist with the selection of test cases.
- Test automation reduces the costs of testing by supporting the testing process with a range of software tools.
- There is not necessarily a simple relationship between the price charged for a system and its development costs. Organizational factors may mean that the price charged is increased to compensate for increased risk or decreased to gain competitive advantage.
- Factors that affect software productivity include individual aptitude (the dominant factor), domain experience, the development process, the size of the project, tool support and the working environment.
- Software is often priced to gain a contract, and the functionality of the system is then adjusted to meet the estimated price.
- There are various techniques of software cost estimation. In preparing an estimate, several different techniques should be used. If the estimates diverge widely, this means that inadequate estimating information is available.
- The COCOMO II costing model is a well-developed algorithmic cost model that takes project, product, hardware and personnel attributes into account when formulating a cost estimate. It also includes a means of estimating development schedules.
- Algorithmic cost models can be used to support quantitative option analysis. They allow the cost of various options to be computed and, even with errors, the options can be compared
- on an objective basis.
- The time required to complete a project is not simply proportional to the number of people working on the project. Adding more people to a late project can increase rather than decrease the time required to finish the project.

16.8 Exercise:

Answer the following:

1. What is the objective of software cost estimation?
2. What are the main costs of a software development project?
3. How programmer productivity can be measured by function points?
4. How programmer productivity can be measured by object points?
5. List some cost estimation techniques that are not based on any size related metric of software?
6. Explain the expression of algorithmic cost modelling?

PROCESS IMPROVEMENT

Unit Structure

- 17.1 Process and Product Quality
 - 17.1.1 Process quality management
- 17.2 Process Classification
- 17.3 Process Measurement
- 17.4 Process Change
- 17.5 The CMMI Process Improvement Framework
 - 17.5.1 The staged CMMI model

17.0 Objectives

In this chapter you will understand What is process and the Quality.

Understand the principles of software process improvement

Understand the cycle process improvement process

Process Improvement:

Process improvement is a cyclical activity. It involves three principal stages:

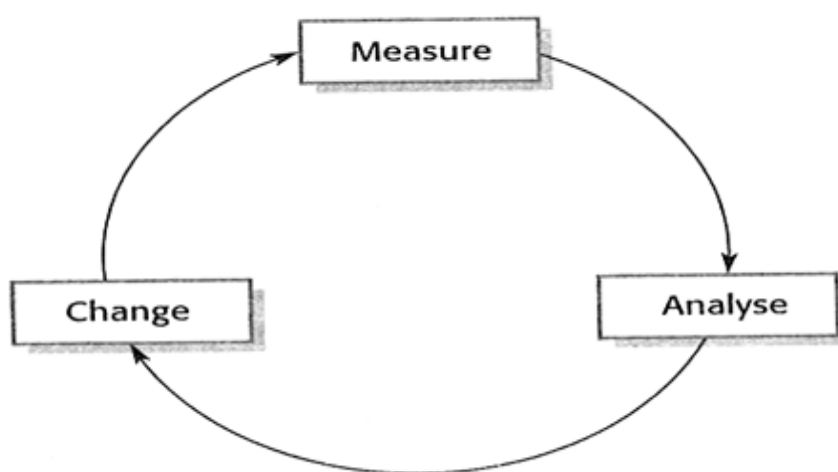


Fig.: The process improvement cycle

- 1) **Process measurement:** Attributes of the current project or the product are measured. The aim is to improve the measures according to the goals of the organization involved in process improvement.

- 2) **Process analysis:** The current process is assessed, and process weaknesses bottlenecks are identified. Process models that describe the process are usually developed during this stage.
- 3) **Process change:** Changes to the process that have been identified during analysis are introduced.

17.1 Process and Product Quality

1. A fundamental assumption of quality management is that the quality of the development process directly affects the quality of delivered products.
2. This assumption comes from manufacturing systems where product quality is intimately related the production process.
3. In an automated manufacturing system, the process involves configuring, setting up and operating the machines involved in process.
4. Once the machines are operating correctly, product quality naturally follows. You measure the quality of the product and change the process until you achieve the quality level that you need.
5. There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor.
6. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products.
7. However, software is not manufactured but is designed.
8. Software development is a creative rather than a mechanical process, so the influence of Individual skills and experience is significant.
9. External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.

17.1.1 Process quality management involves:

- 1) Defining process standards such as how and when reviews should be conducted
- 2) Monitoring the development process to ensure that the standards are being followed
- 3) Reporting the software process to project management and to the buyer of the software

One problem with process-based quality assurance is that the quality assurance (QA) team may insist that standard processes should be used irrespective of the type of software that is being developed.

For example, process quality standards for critical systems may specify that specification must be complete and approved before implementation can begin.

However, some critical systems may require prototyping where programs are implemented without a complete specification.

I have experienced situations where the quality management team suggests that this prototyping should not be carried out because the prototype quality cannot be monitored.

In such situations, senior management must intervene to ensure that the quality process supports rather than hinders product development.

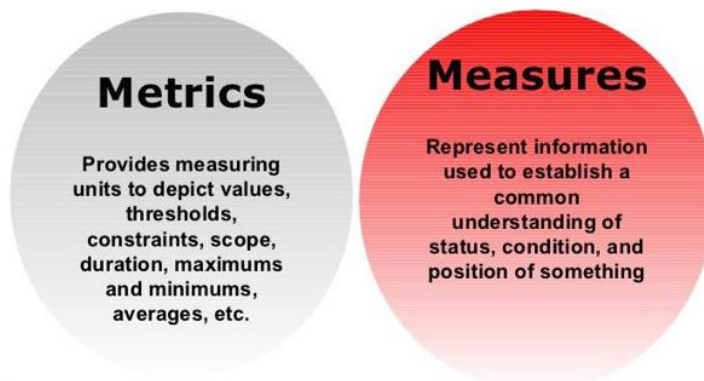


Fig: Metrics and Measures

Process characteristic	Description
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition?
Visibility	Do the process activities culminate in clear results so that the progress of the process is externally visible?
Supportability	To what extent can CASE tools be used to support the process activities?
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?
Maintainability	Can the process evolve to reflect changing organizational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

Fig: Process characteristics

17.2 Process Classification

- 1) **Informal processes.** When there is no strictly defined process model, the development team chooses the process that they will use. Informal processes may use formal procedures such as configuration management, but the procedures and the relationships between procedures are defined as required by the development team.

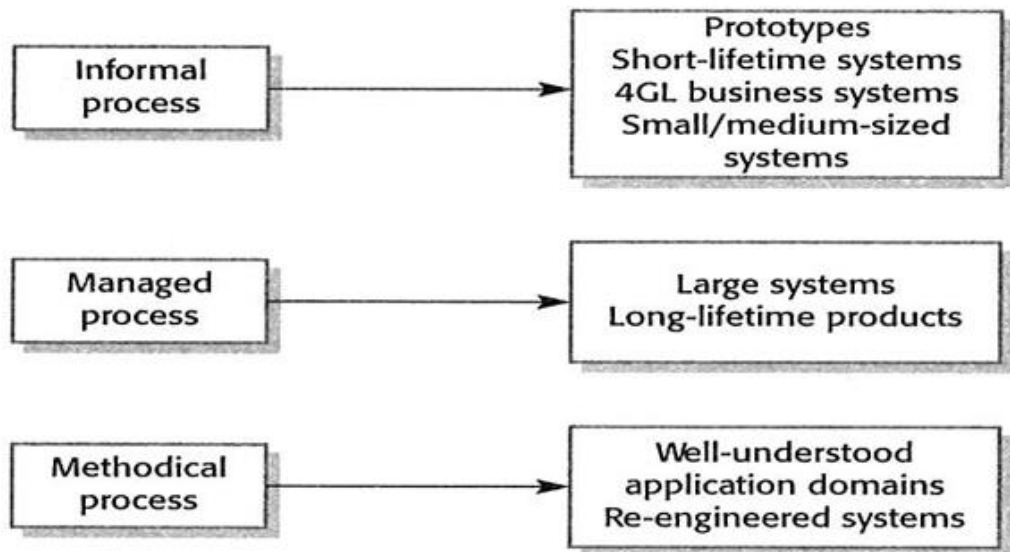


Fig.: Process applicability

- 2) **Managed processes.** A defined process model is used to drive the development process. The process model defines the procedures, their scheduling, and the relationships between the procedures.
- 3) **Methodical processes.** When some defined development method are used, these processes benefit from CASE tool support for design and analysis processes.
- 4) **Improving processes.** Processes that have inherent improvement objectives have a specific budget for improvements and procedures for introducing SUCH improvements. As part of this, quantitative process measurement may be introduced.

17.3 Process Measurement

Process measurements are quantitative data about the software process.

- The measurement of process and product attributes is essential for process improvement.
- Measurement has an important role to play in small-scale, personal process improvement.

- Process measurements can be used to assess whether the efficiency of a process has been improved.
- For example, the effort and time devoted to testing can be monitored.
- Effective improvements to the testing process should reduce the effort, testing time or both.
- However, process measurements on their own cannot be used to determine whether product quality has improved.
- Product quality data should also be collected and related to the process activities

Three classes of process metric can be collected:

- 1) **The time taken for a process to be completed.** This can be the total time devoted to the process, calendar time, the time spent on the process by engineers.
- 2) **The resources required for a process.** The resources might include total effort in person-days, travel costs and computer resources.
- 3) **The number of occurrences of an event.** Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested and the average number of lines of code modified in response to a requirement change.

This approach relies on the identification of:

- 1) Goals. What the organization is trying to achieve. Examples of goals might be improved programmer productivity, shorter product development time and increased product reliability.
- 2) Questions. These are refinements of goals where specific areas of uncertainty related to the goals are identified. Normally, a goal will have several associated questions that need to be answered. Examples of questions related to the above goals are:
 - How can the number of debugged lines of code be increased?
 - How can the time required to finalize product requirements be reduced?

How can more effective reliability assessments be made?

- 3) Metrics. These are the measurements that need to be collected to help answer the questions and to confirm whether process improvements have achieved the desired goal.

Advantages

- 1) Separate organizational concerns from specific process concerns.
- 2) focus on data collection and suggest the data to be analyse in different ways depending on question planned to answer.

17.4 Process Change

Process change involves making modifications to the existing process. You may do this by introducing new practices, methods, or tools, by changing the ordering of process activities, by introducing or removing deliverables from the process, or by introducing new roles and responsibilities.

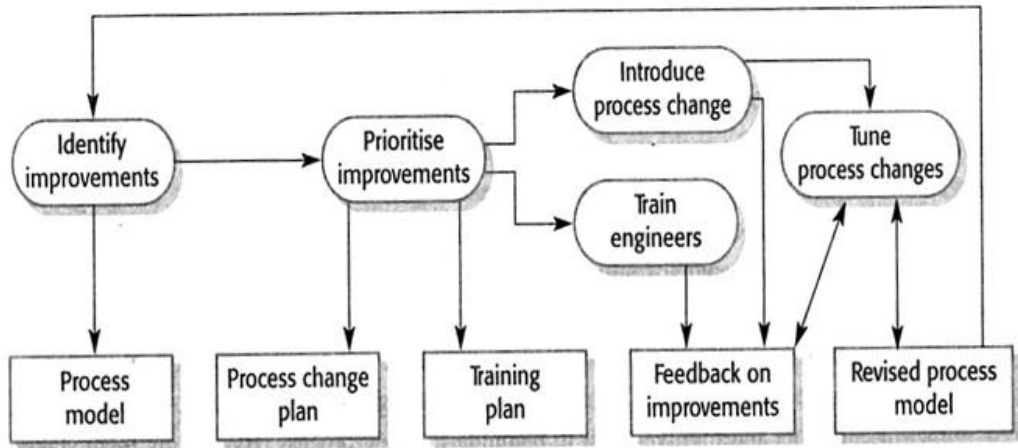


Fig.: The process change process

There are five key stages in the process change process:

- 1) **Improvement identification.** This stage is concerned with using the results of the process analysis to identify quality, schedule, or cost bottlenecks where process factors might adversely influence the product quality
- 2) **Improvement prioritizations.** This stage is concerned with assessing the possible changes and prioritizing them for implementation.
Which are most important. You may make these decisions based on the need to improve specific process areas, the costs of introducing the change, the impact of the change on the organization and other factors.
- 3) **Process change's introduction.** Process change introduction means putting new procedures, methods, and tools into place, and integrating them with other process activities. You must allow enough time to introduce changes and to ensure that these changes are compatible with other process activities and with organizational procedures and standards.
- 4) **Process change training.** Without training, it is not possible to gain the full benefits from process changes. Process managers and software engineers may simply refuse to accept the new process.
- 5) **Change tuning.** Proposed process changes will never be completely effective as soon as they are introduced. You need a tuning phase where minor problems are discovered, and modifications to the process are proposed and are introduced.

17.5 The CMMI Process Improvement Framework

The CMMI model is intended to be a framework for process improvement that has broad applicability across a range of companies.

Its staged version is compatible with the Software CMM and allows an organization's system development and management processes to be assessed and assigned a maturity level from 1 to 5.

Category	Process area
Process management	Organizational process definition Organizational process focus Organizational training Organizational process performance Organizational innovation and deployment
Project management	Project planning Project monitoring and control Supplier agreement management Integrated project management Risk management Integrated teaming Quantitative project management
Engineering	Requirements management Requirement's development Technical solution Product integration Verification Validation
Support	Configuration management Process and product quality management Measurement and analysis Decision analysis and resolution Organizational environment for integration Causal analysis and resolution

Fig: Process areas in the CMMI

- 1) **Process areas.** The CMMI identifies 24 process areas that are relevant to software process capability and improvement. These are organized into four groups in the continuous CMMI model. These groups and related process areas are listed above.
- 2) **Goals.** Goals are abstract descriptions of a desirable state that should be attained by an organization. The CMMI has specific goals that are associated with each process area and that define the desirable state for that area. It also defines generic goals that are associated with the institutionalization of good practice.

- 3) **Practices.** Practices in the CMMI are descriptions of ways to achieving a goal. Up to seven specific and generic practices may be associated with each goal within each process area.

Goal	Process Area
Corrective actions are managed to closure when the project's performance or results deviate significantly from the plan	Specific goal in project monitoring and control
Actual performance and progress of the project is monitored against the project plan	Specific goal in project monitoring and control
The requirements are analyzed and validated, and a definition of the required functionality is developed	Specific goal in requirements development
Root causes of defects and other problems are systematically determined	Specific goal in causal analysis and resolution
The process is institutionalized as a defined process	Generic goal

Fig: Process area in the CMMI

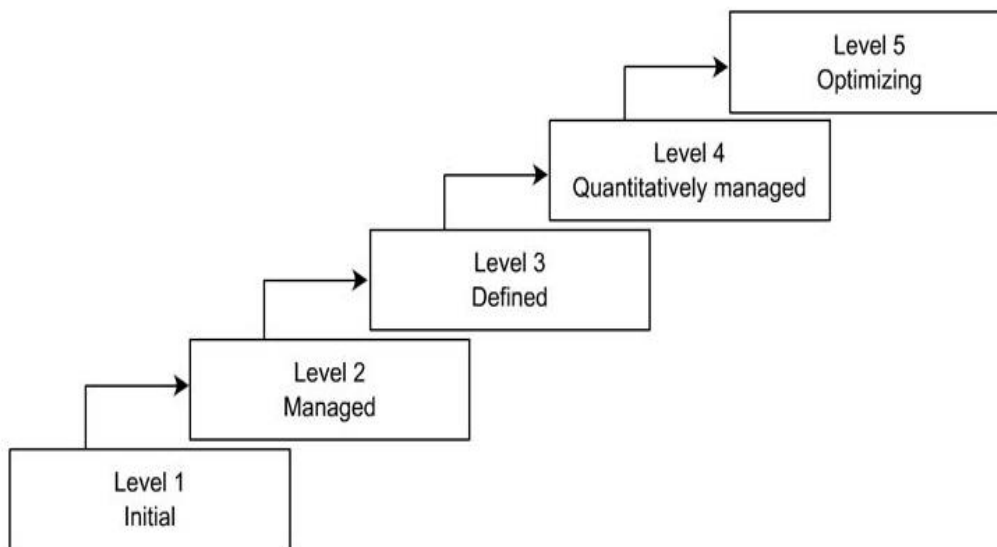
A CMMI assessment involves examining the processes in an organization and rating these on a six-point scale that relates to the level of maturity in each process area.

The six-point scale assigns a level to a process as follows:

1. **Incomplete:** At least one of the specific goals associated with the process area is not satisfied. There are no generic goals at this level as institutionalization of an incomplete process does not make sense.
2. **Performed:** The goals associated with the process area are satisfied, and for all processes the scope of the work to be performed is explicitly set out and communicated to the team members.
3. **Managed:** At this level, the goals associated with the process area are met and organizational policies are in place that define when each process should be used. There must be documented project plans that define the project goals. Resource management and process monitoring procedures must be in place across the institution.
4. **Defined:** This level focuses on organizational standardization and deployment of processes. Each project has a managed process that is adapted to the project requirements from a defined set of organizational processes. Process assets and process measurements must be collected and used for future process improvements.

5. **Quantitatively:** managed at this level, there is an organizational responsibility to use statistical and other quantitative methods to control subprocesses; that is, collected process and product measurements must be used in process management.
6. **Optimizing:** At this highest level, the organization must use the process and product measurements to drive process improvement. Trends must be analyzed, and the processes adapted to changing business needs.

17.5.1 The staged CMMI model



1. **Requirements management** Manage the requirements of the project's products and product components and identify inconsistencies between those requirements and the project's plans and work products.
2. **Project planning** Establish and maintain plans that define project activities.
3. **Project monitoring and control** Provide understanding into the project's progress so that appropriate corrective actions can be taken when the project's performance deviates significantly from the plan.
4. **Supplier agreement management** Manage the acquisition of products and services from suppliers external to the project for which a formal agreement exists.
5. **Measurement and analysis** Develop and sustain a measurement capability that is used to support management information needs.
6. **Process and product quality assurance** Provide staff and management with objective insight into the processes and associated work products.

- 7. Configuration management** Establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits.

Graded Question

1. Write a note on ISO 9000 quality standards.
2. Explain process improvement cycle.
3. Explain CMMI framework.
4. Explain Software as service.
5. Explain service engineering with neat diagram.

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

SERVICE ORIENTED SOFTWARE ENGINEERING

Unit Structure

- 18.0 Objectives
- 18.1 Introduction
- 18.3 Contents
 - 18.3.1 Overview of Service-Oriented Architecture (SOA)
 - 18.3.2 Services as Reusable Components
 - 18.3.3 Service Engineering
 - 18.3.4 Software Development with Services
- 18.4 Key Points
- 18.5 Bibliography
- 18.6 Exercises

18.0 Objectives

Service-Oriented Software Engineering (SOSE) is a software engineering methodology focused on the development of software systems by composition of reusable services (service-orientation) often provided by other service providers. The main aim of this chapter is to introduce service - oriented software architecture as a way of building distributed applications using web services.

18.1 Introduction

Service-Oriented Architecture (SOA) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. Its principles are independent of vendors and other technologies. In service-oriented architecture, a number of services communicate with each other, in one of two ways: through passing data or through two or more services coordinating an activity. This is just one definition of Service-Oriented Architecture.

18.3 Contents

18.3.1 Overview of Service-Oriented Architecture (Soa)

- In 1990s the web development mainly used for information exchange between organization.

- The environment architecture as Client-Server.
- The client send request to the server to access information on the server system that located outside of their own organizations.
- The web browser is responsible for directing client access to information on the server.

Disadvantages

- For example, a program queried several catalogs from different suppliers, were not possible.

Solution – Web Service

- The notion of a web service was proposed.
- A web service is a standard representation for some computational or information resource that can be used by other programs.

Definition of Web Service

An act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production.

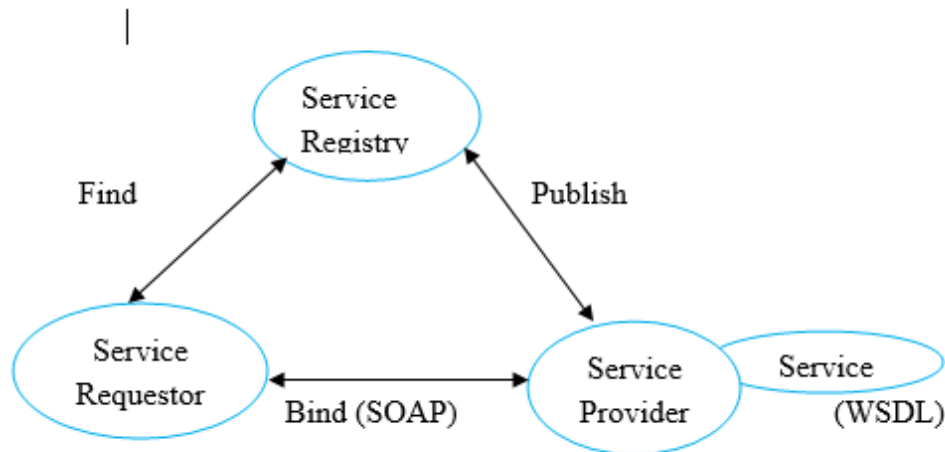


Figure 18.1. Service-Oriented Architecture

The Figure 18.1, shows the stack of key standards that have been established to support web services. Web service protocols cover all aspects of SOAs, from the basic mechanisms for service information exchange (SOAP) to programming language standards (WS-BPEL).

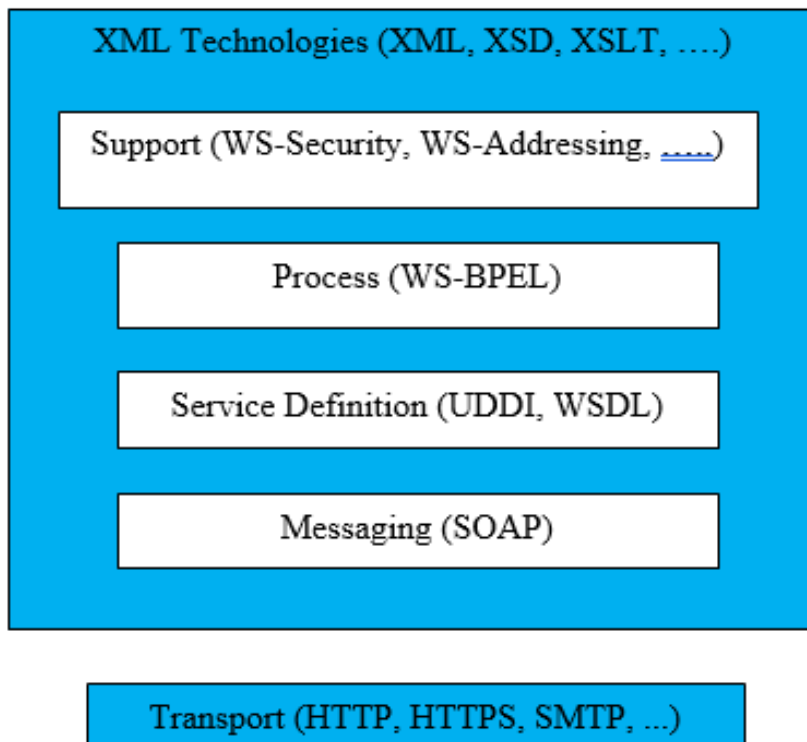
All these standards based on XML, human and machine-readable notation that allows the definition of structured data where text is tagged with a meaningful identifier.

- XML has a range of supporting technologies such as XSD.
- XSD – for schema definition which are used to extend and manipulate XML descriptions.

The key standards for web SOAs

- **SOAP** – This is a message interchange standard that supports the communication between services. It defines the essential and optional components of messages passed between services.
- **WSDL** – It is a standard for service interface definition. It sets out how the service operations (operation names, parameters, and their types) and service bindings should be defined.
- **WS-BPEL** - This is a standard for a workflow language that is used to define process programs involving several different services

Figure 18.2. Web Service Standards



Examples of Web Service Standards include the following:

- **WS-Reliable Messaging**
A standard for message exchange that ensures messages will be delivered once and once only.

- **WS-Security**

A set of standards supporting web service security including standards that specify the definition of security policies and standards that cover the use of digital signatures.

- **WS-Addressing**

This defines how address information should be represented in a SOAP message.

- **WS-Transactions**

This defines how transactions across distributed services should be coordinated.

18.3.2 Services As Reusable Components

A service can be defined as the following:

A loosely-coupled, reusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML- based protocols.

Services

- The services should be independent and loosely coupled; that is, they should always operate in the same way, irrespective of their execution environment.
- Their interface is a ‘provides’ interface that allows access to the service functionality.
- Services are intended to be independent and usable in different contexts.
- It communicates by exchanging messages, expressed in XML, and these messages are distributed using standard Internet transport protocols such as HTTP and TCP/IP.
- A service defines what it needs from another service by setting out its requirements in a message and sending it to that service.
- The receiving service parses the message, carries out the computation and, on completion, sends a reply, as a message, to the requesting service.
- This service then parses the reply to extract the required information. Unlike software components, services do not use remote procedure or method calls to access functionality associated with other services.
- Whenever we use a web service, we want to know the URI of where the service is located.
- These are described in a service description expressed in an XML-based language called WSDL.

The WSDL specification defines three things about a web service:

- **what the service does**

The ‘**what**’ part of a WSDL document, called an interface, specifies what operations the service supports, and defines the format of the messages that are sent and received by the service?

- **how it communicates**

The ‘**how**’ part of a WSDL document, called a binding, maps the abstract interface to a concrete set of protocols. The binding specifies the technical details of how to communicate with a web service.

- **where to find it**

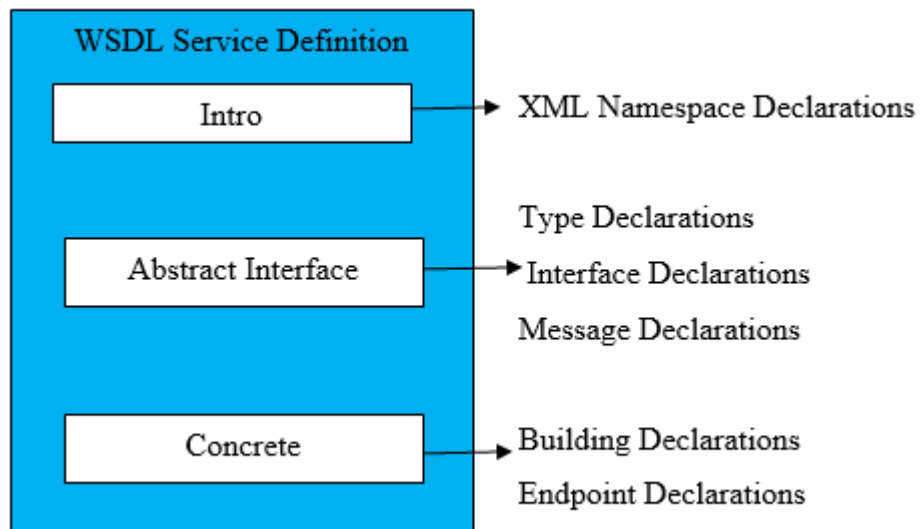
The ‘**where**’ part of a WSDL document describes the location of a specific web service implementation (its endpoint).

WSDL conceptual Model:

It shows the elements of a service description. Each of these is expressed in XML and may be provided in separate files.

These parts are:

1. An introductory part that usually defines the XML namespaces used and which may include a documentation section providing additional information about the service.
2. An optional description of the types used in the messages exchanged by the service.
3. A description of the service interface; that is, the operations that the service provides for other services or users.
4. A description of the input and output messages processed by the service.
5. A description of the binding used by the service (i.e., the messaging protocol that will be used to send and receive messages). The default is SOAP but other bindings may also be specified. The binding sets out how the input and output messages associated with the service should be packaged into a message, and specifies the communication protocols used. The binding may also specify how supporting information, such as security credentials or transaction identifiers, is included.
6. An endpoint specification which is the physical location of the service, expressed as a Uniform Resource Identifier (URI)—the address of a resource that can be accessed over the Internet.

Figure 18.3. Organization of a WSDL specification

- Complete service descriptions, written in XML, are long, detailed, and tedious to read.
- They usually include definitions of XML namespaces, which are qualifiers for names. A namespace identifier may precede any identifier used in the XML description, making it possible to distinguish between identifiers with the same name that have been defined in different parts of an XML description.
- WSDL specifications are now rarely written by hand and most of the information in a specification can be automatically generated.

18.3.3 Service Engineering

- a. Introduction
- b. Stages
 - i) Service Candidate Identification
 - ii) Service Interface Design
 - iii) Service Implementation and Deployment
- c. Legacy System Services

a. Introduction

- Service engineering is the process of developing services for reuse in service-oriented applications.
- It has much in common with component engineering.

- Service engineers have to ensure that the service represents a reusable abstraction that could be useful in different systems.
- They must design and develop generally useful functionality associated with that abstraction and ensure that the service is robust and reliable.
- They have to document the service so that it can be discovered and understood by potential users.

b. Three Logical stages in the Service Engineering Process

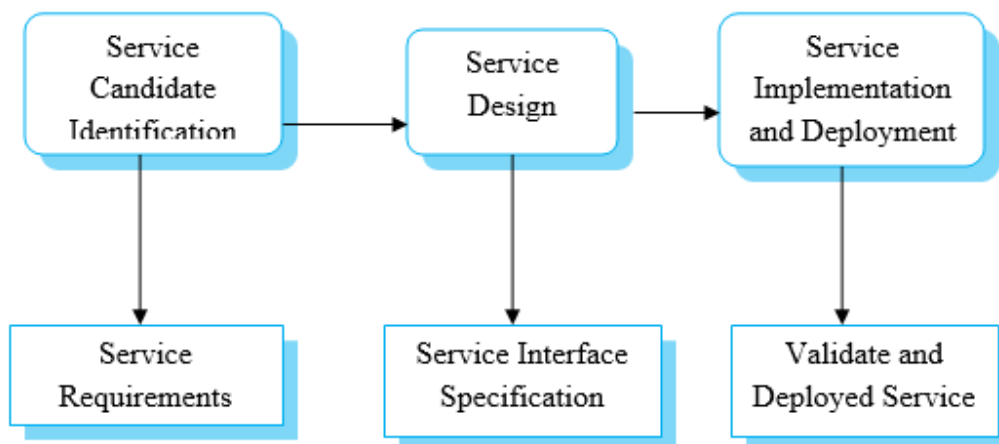
- i) Service candidate identification - where you identify possible services that might be implemented and define the service requirements.
- ii) Service design - where you design the logical and WSDL service interfaces.
- iii) Service implementation and deployment - where you implement and test the service and make it available for use.

i) Service Candidate Identification

Basic Notion

- Services should support business processes
- As every organization has a wide range of processes
- Service candidate identification therefore involves understanding and analyzing the organization's business processes to decide which reusable services could be implemented to support these processes.

Figure 18.4 The Service Engineering Process



Three fundamental types of Service

- i) **Utility Services** - These are services that implement some general functionality that may be used by different business processes. An example of a utility service is a currency conversion service that can be accessed to compute the conversion of one currency (e.g., dollars) to another (e.g., euros).
- ii) **Business Services** - These are services that are associated with a specific business function. An example of a business function in a university would be the registration of students for a course.
- iii) **Coordination or Process Services** - These are services that support a more general business process which usually involves different actors and activities. An example of a coordination service in a company is an ordering service that allows orders to be placed with suppliers, goods accepted, and payments made.

Table 18.1 Service Classification

	Utility	Business	Coordination
Task	Currency converter Employee locator	Validate claim form Check credit rating	Process expense claim Pay external supplier
Entity	Document style checker Web form to XML converter	Expenses form Student application form	-

Example – Catalog Service

The catalog service is an example of an entity-oriented service that supports business operations. **The functional catalog service requirements are as follows:**

- 1. A specific version of the catalog shall be provided for each user company. This shall include the configurations and equipment that may be ordered by employees of the customer company and the agreed prices for catalog items.
- 2. The catalog shall allow a customer employee to download a version of the catalog for offline browsing.
- 3. The catalog shall allow users to compare the specifications and prices of up to six catalog items.

4. The catalog shall provide browsing and search facilities for users.
5. Users of the catalog shall be able to discover the predicted delivery date for a given number of specific catalog items.
6. Users of the catalog shall be able to place ‘virtual orders’ where the items required will be reserved for them for 48 hours. Virtual orders must be confirmed by a real order placed by a procurement system. This must be received within 48 hours of the virtual order.

In addition to these functional requirements, the catalog has a number of nonfunctional requirements:

1. Access to the catalog service shall be restricted to employees of accredited organizations.
2. The prices and configurations offered to one customer shall be confidential and shall not be available to employees of any other customer.
3. The catalog shall be available without disruption of service from 0700 GMT to 1100 GMT.
4. The catalog service shall be able to process up to 10 requests per second peak load.

ii) Service Interface Design

Once you have selected candidate services, the next stage in the service engineering process is to design the service interfaces. This involves defining the operations associated with the service and their parameters.

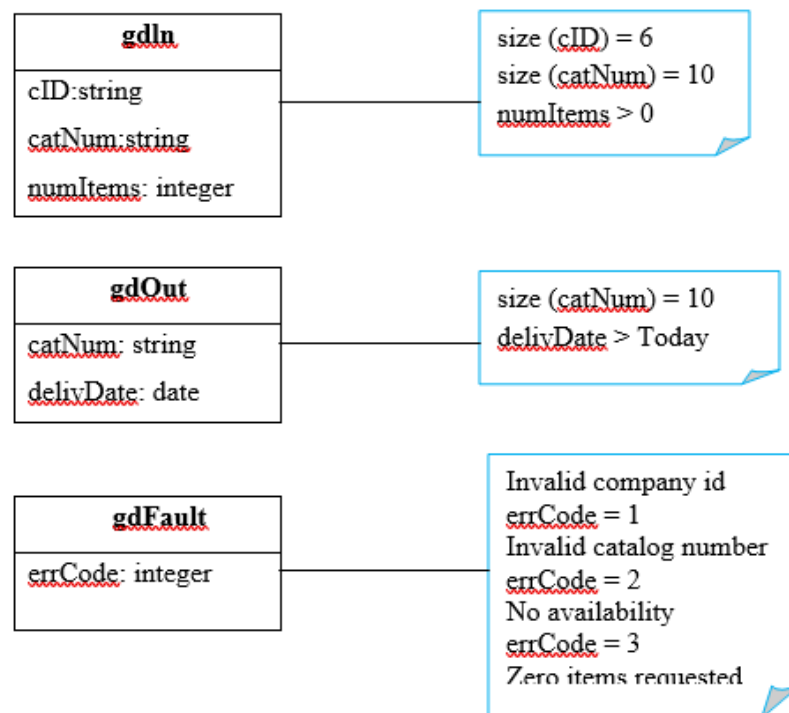
Table 18.2. Functional Descriptions of Catalog Service Operations

Operation	Description
Make Catalog	Creates a version of the catalog tailored for a specific customer. Includes an optional parameter to create a downloadable PDF version of the catalog.
Compare	Provides a comparison of up to six characteristics (e.g., price, dimensions, processor speed, etc.) of up to four catalog items.
Lookup	Displays all of the data associated with a specified catalog item.
Search	This operation takes a logical expression and searches the catalog according to that expression.
Check Delivery	Returns the predicted delivery date for an item if ordered that day.
Make Virtual Order	Reserves the number of items to be ordered by a customer and provides item information for the customer’s own procurement system

There are three stages to service interface design:

1. **Logical interface design** - where you identify the operations associated with the service, their inputs and outputs and the exceptions associated with these operations.
 - Starts with the service requirements and defines the operation names and parameters. At this stage, you should also define the exceptions that may arise when a service operation is invoked.
2. **Message design** - where you design the structure of the messages that are sent and received by the service.
3. **WSDL development** - where you translate your logical and message design to an abstract interface description written in WSDL.

Figure 18.5. Functional Descriptions of Catalog Service Operations



- Figure 18.5, and Table 18.3, show the operations that implement the requirements and the inputs, outputs, and exceptions for each of the catalog operations. At this stage, there is no need for these to be specified in detail — you add detail at the next stage of the design process.
- Defining exceptions and how these can be communicated to service users is particularly important. Service engineers do not know how their services will be used.

- Define Exceptions, if Input messages may be incorrect and that report incorrect inputs to the service client.
- Once you have established an informal logical description of what the service should do, the next stage is to define the structure of the input and output messages and the types used in these messages.
- XML is an awkward notation to use at this stage.

Table 18.3. UML Definition of Input and Output Messages

Operation	Inputs	Outputs	Exceptions
Make Catalog	mcIn Company id PDF-flag	mcOut URL of the catalog for that company	mcFault Invalid company id
Compare	compIn Company id Entry attribute (up to 6) Catalog number (up to 4)	compOut URL of page showing comparison table	compFault Invalid company id Invalid catalog number Unknown attribute
Lookup	lookIn Company id Catalog number	lookOut URL of page with the item information	lookFault Invalid company id Invalid catalog number
Search	searchIn Company id Search string	searchOut URL of web page with search results	searchFault Invalid company id Badly formed search string
Check Delivery	gdIn Company id Catalog number Number of items required	gdOut Catalog number Expected delivery date	gdFault Invalid company id Invalid catalog number No availability Zero items requested
Place Order	poIn Company id Number of items required Catalog number	poOut Catalog number Number of items required Predicted delivery date Unit price estimate Total price estimate	poFault Invalid company id Invalid catalog number Zero items requested

iii) Service Implementation and Deployment

- Once you have identified candidate services and designed their interfaces, the final stage of the service engineering process is service implementation.
- This implementation may involve programming the service using a standard programming language such as Java or C#.
- Both of these languages include libraries with extensive support for service development.
- Alternatively, services may be developed by implementing service interfaces to existing components.
- Once a service has been implemented, it then has to be tested before it is deployed.
- This involves examining and partitioning the service inputs.
- Creating input messages that reflect these input combinations, and then checking that the outputs are expected.
- Testing tools are available that allow services to be examined and tested, and that generate tests from a WSDL specification.
- Service deployment, the final stage of the process, involves making the service available for use on a web server.
- Most server software makes this very simple.
- You only have to install the file containing the executable service in a specific directory.
- If the service is intended to be publicly available, you then have to provide information for external users of the service.
- This information helps potential external users to decide if the service is likely to meet their needs and if they can trust you, as a service provider, to deliver the service reliably and securely.

Included Information in a Service Description:

- i) Information about your business, contact details, etc. This is important for trust reasons. Users of a service have to be confident that it will not behave maliciously. Information about the service provider allows them to check their credentials with business information agencies.
- ii) An informal description of the functionality provided by the service. This helps

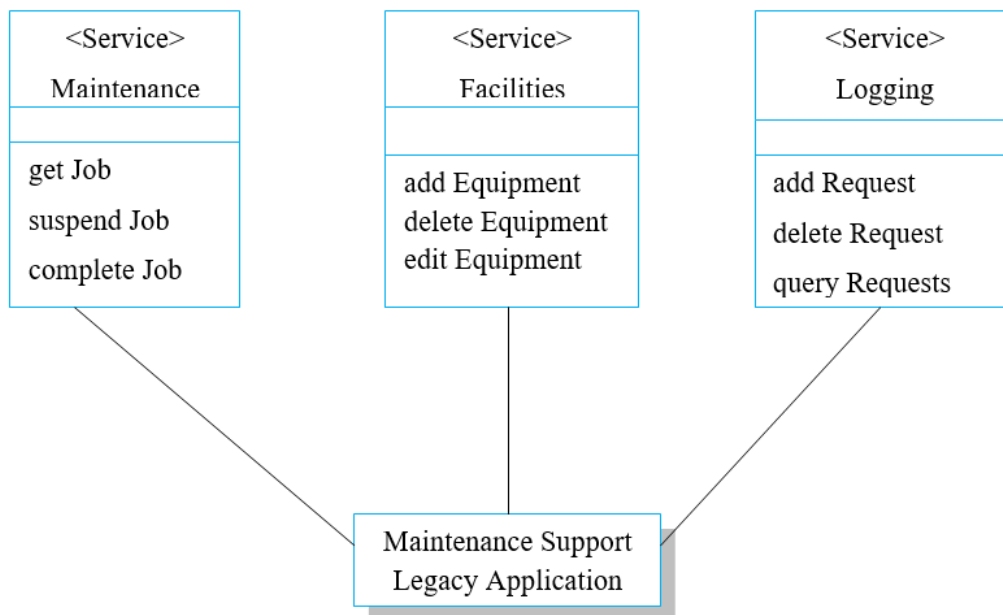
potential users to decide if the service is what they want. However, the functional description is in natural language, so it is not an unambiguous semantic description of what the service does.

- iii) A detailed description of the interface types and semantics.
- iv) Subscription information that allows users to register for information about updates to the service.

c. Legacy System Services

Legacy systems are old software systems that are used by an organization. Usually, they rely on obsolete technology but are still essential to the business. It may not be cost effective to rewrite or replace these systems and many organizations would like to use them in conjunction with more modern systems. One of the most important uses of services is to implement ‘wrappers’ for legacy systems that provide access to a system’s functions and data. These systems can then be accessed over the Web and integrated with other applications.

Figure 18.6. Services providing access to a legacy system



Some of the services provided are the following:

- i) **A maintenance service** - This includes operations to retrieve a maintenance job according to its job number, priority, and geographical location, and to upload details of maintenance that has been carried out to the maintenance database. The service also provides operations that allow a maintenance job that has started but is incomplete to be suspended and restarted.

- ii) **A facilities service** - This includes operations to add and delete new equipment and to modify the information associated with equipment in the database.
- iii) **A logging service** - This includes operations to add a new request for service, delete maintenance requests, and query the status of outstanding requests.

18.3.4 Software Development with Services

- a. **Introduction**
- b. **Workflow Design and Implementation**
- c. **Service Testing**

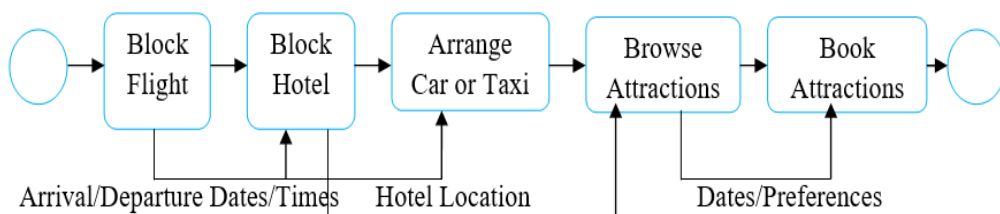
a. Introduction

The development of software using services is based around the idea that you compose and configure services to create new, composite services. These may be integrated with a user interface implemented in a browser to create a web application, or may be used as components in some other service composition. The services involved in the composition may be specially developed for the application, may be business services developed within a company, or may be services from an external provider. Service composition may be used to integrate separate business processes to provide an integrated process offering more extensive functionality.

Example: Airline Reservation System

- An airline wishes to provide a complete vacation package for travelers.
- As well as booking their flights, travelers can also book hotels in their preferred location, arrange car rentals or book a taxi from the airport, browse a travel guide, and make reservations to visit local attractions.
- To create this application, the airline composes its own booking service with services offered by a hotel booking agency, car rental and taxi companies, and reservation services offered by owners of local attractions.
- The end result is a single service that integrates the services from different providers.

Figure 18.7. Vacation Package Workflow

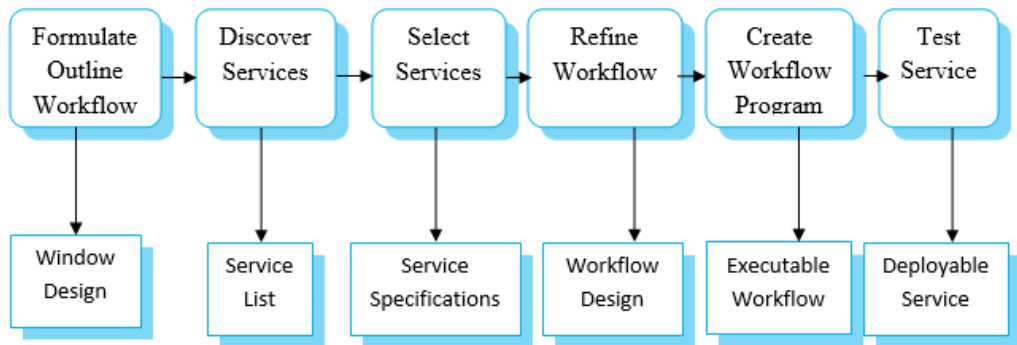


- A sequence of separate steps as shown in Figure 18.7. Information is passed from one step to the next—for example; the car rental company is informed of the time that the flight is scheduled to arrive.
- **The sequence of steps is called a workflow**—a set of activities ordered in time, with each activity carrying out some part of the work.
- **A workflow is a model of a business process** (i.e., sets out the steps involved in reaching a particular goal that is important for a business).
- In this case, the business process is the vacation booking service, offered by the airline.
- Workflow is a simple idea and the above scenario of booking a vacation seems to be straightforward.
- In practice, service composition is much more complex than this simple model implies.
- For example, you have to consider the possibility of service failure and incorporate mechanisms to handle these failures.
- You also have to take into account exceptional demands made by users of the application.
- For example, say a traveler was disabled and required a wheelchair to be rented and delivered to the airport.
- This would require extra services to be implemented and composed, and additional steps to be added to the workflow.

The process of designing new services by reusing existing services is essentially a process of software design with reuse (Figure 18.8). Design with reuse inevitably involves requirements compromises. The ‘ideal’ requirements for the system have to be modified to reflect the services that are actually available, whose costs fall within budget and whose quality of service is acceptable.

In Figure 18.8. shown six key stages in the process of service construction by composition:

1. Formulate outline workflow - In this initial stage of service design, you use the requirements for the composite service as a basis for creating an ‘ideal’ service design. You should create a fairly abstract design at this stage with the intention of adding details once you know more about available services.

Figure 18.8. Service Construction by Composition

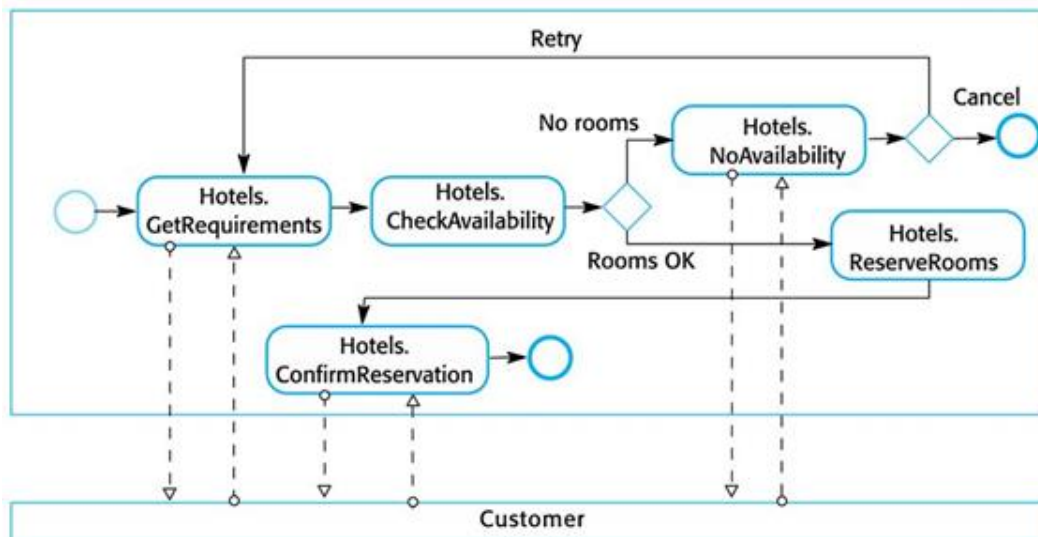
2. **Discover services** - During this stage of the process, you search service registries or catalogs to discover what services exist, who provides these services, and the details of the service provision.
3. **Select possible services** - From the set of possible service candidates that you have discovered, you then select possible services that can implement workflow activities. Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered. You may decide to choose a number of functionally equivalent services, which could be bound to a workflow activity depending on details of cost and quality of service.
4. **Refine workflow** - On the basis of information about the services that you have selected, you then refine the workflow. This involves adding detail to the abstract description and perhaps adding or removing workflow activities. You may then repeat the service discovery and selection stages. Once a stable set of services has been chosen and the final workflow design established, you move on to the next stage in the process.
5. **Create workflow program** - During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can use a conventional programming language, such as Java or C#, for service implementation or a workflow language, such as WS-BPEL. The service interface specification should be written in WSDL. This stage may also involve the creation of web-based user interfaces to allow the new service to be accessed from a web browser.
6. **Test completed service or application** - The process of testing the completed,

composite service is more complex than component testing in situations where external services are used.

b. Workflow Design and Implementation

Workflow design involves analyzing existing or planned business processes to understand the different activities that go on and how this exchange information.

Figure 18.9. A Fragment of a Hotel Booking Workflow



You then define the new business process in a workflow design notation. This sets out the stages involved in enacting the process and the information that is passed between the different process stages.

However, existing processes may be informal and dependent on the skills and ability of the people involved there may be no ‘normal’ way of working or process definition. In such cases, you have to use your knowledge of the current process to design a workflow that achieves the same goals.

Workflows represent business process models and are usually represented using a graphical notation such as UML activity diagrams or BPMN, the Business Process Modeling Notation (White, 2004a; White and Miers, 2008). These offer similar features (White, 2004b). Mappings have been defined to translate the language to lower-level, XML-based descriptions in WS-BPEL.

Figure 18.9, is an example of a simple BPMN model of part of the above vacation package scenario. The model shows a simplified workflow for hotel booking and assumes the existence of a Hotels service with associated operations called GetRequirements, CheckAvailability, ReserveRooms, NoAvailability, Confirm Reservation, and CancelReservation. The process involves getting requirements

from the customer, checking room availability, and then, if rooms are available, making a booking for the required dates.

This model introduces some of the core concepts of BPMN that are used to create workflow models:

1. Activities are represented by a rectangle with rounded corners. An activity can be executed by a human or by an automated service.
2. Events are represented by circles. An event is something that happens during a business process. A simple circle is used to represent a starting event and a darker circle to represent an end event. A double circle (not shown) is used to represent an intermediate event. Events can be clock events, thus allowing workflows to be executed periodically or timed out.
3. A diamond is used to represent a gateway. A gateway is a stage in the process where some choice is made. For example, in Figure 18.9, there is a choice made on the basis of whether rooms are available or not.
4. A solid arrow is used to show the sequence of activities; a dashed arrow represents message flow between activities. In Figure 18.9, these messages are passed between the hotel booking service and the customer.

c. Service Testing

Testing is important in all system development processes as it demonstrates that a system meets its functional and non-functional requirements and to detect defects that have been introduced during the development process. Many testing techniques,

such as program inspections and coverage testing, rely on analysis of the software source code. However, when services are offered by an external provider, source code of the service implementation is not available. Service-based system testing cannot therefore use proven source code-based techniques.

As well as problems of understanding the implementation of the service, testers may also face further difficulties when testing services and service compositions:

1. External services are under the control of the service provider rather than the user of the service. The service provider may withdraw these services at any time or may make changes to them, which invalidates any previous application testing. These problems are handled in software components by maintaining different versions of the component. Currently, however, there are no standards proposed to deal with service versions.
2. The long-term vision of SOAs is for services to be bound dynamically to service-oriented applications. This means that an application may not

always use the same service each time that it is executed. Therefore, tests may be successful when an application is bound to a particular service, but it cannot be guaranteed that that service will be used during an actual execution of the system.

3. The non-functional behavior of a service is not simply dependent on how it is used by the application that is being tested. A service may perform well during testing because it is not operating under a heavy load. In practice, the observed service behavior may be different because of the demands made by other service users.
4. The payment model for services could make service testing very expensive. There are different possible payment models some services may be freely available, some paid for by subscription, and others paid for on a per-use basis.

If services are free, then the service provider will not wish them to be loaded by applications being tested; if a subscription is required, then a service user may be reluctant to enter into a subscription agreement before testing the service. Similarly, if the usage is based on payment for each use, service users may find the cost of testing to be prohibitive.

5. We discussed the notion of compensation actions that are invoked when an exception occurs and previous commitments that have been made (such as a flight reservation) have to be revoked. There is a problem in testing such actions as they may depend on the failure of other services. Ensuring that these services actually fail during the testing process may be very difficult.

18.4 Key Points

- Service-oriented architecture is an approach to software engineering where reusable, standardized services are the basic building blocks for application systems. Service interfaces may be defined in an XML-based language called WSDL.
- A WSDL specification includes a definition of the interface types and operations, the binding protocol used by the service and the service location.
- Services may be classified as utility services that provide a general-purpose functionality, business services that implement part of a business process, or coordination services that coordinate the execution of other services.

- The service engineering process involves identifying candidate services for implementation, defining the service interface and implementing, and testing and deploying the service.
- Service interfaces may be defined for legacy software systems that continue to be useful for an organization. The functionality of the legacy system may then be reused in other applications.
- The development of software using services is based around the idea that programs are created by composing and configuring services to create new composite services.
Business process models define the activities and information exchange that takes place in a business process.
- Activities in the business process may be implemented by services so that the business process model represents a service composition.

18.5 Bibliography

1. **‘Software Engineering’**, Ian Somerville, Pearson Education. Ninth Edition.
2. **‘Software Engineering’**, Roger Pressman, Tata-Mcgraw Hill, Seventh Edition.
3. Andrews, T., Curbera, F., Golland, Y., Klein, J. and Al., E. (2003). **‘Business Process Execution Language for Web Services’**.
<http://www-128.ibm.com/developerworks/library/ws-bpel/>.
4. Cabrera, L. F., Copeland, G. and Al., E. 2005. **‘Web Services Coordination (WS-Coordination)’**.
<ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf>.
5. Carr, N. (2009). **‘The Big Switch: Rewiring the World from Edison to Google’**, Reprint edition. New York: W.W. Norton & Co.
6. Erl, T. (2004). **‘Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services’**, Upper Saddle River, NJ: Prentice Hall.
7. Erl, T. (2005). **‘Service-Oriented Architecture: Concepts, Technology and Design’**, Upper Saddle River, NJ: Prentice Hall.
8. Kavantzaz, N., Burdett, D. and Ritzinger, G. 2004. **‘Web Services Choreography Description Language Version 1.0’**.
<http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>.

9. Lovelock, C., Vandermerwe, S. and Lewis, B. (1996). **‘Services Marketing’**, Englewood Cliffs, NJ: Prentice Hall.
10. Newcomer, E. and Lomow, G. (2005). **‘Understanding SOA with Web Services’**, Boston: Addison-Wesley.
11. Owl_Services_Coalition. 2003. **‘OWL-S: Semantic Markup for Web Services’**.
<http://www.daml.org/services/owl-s/1.0/owl-s.pdf>.
12. Pautasso, C., Zimmermann, O. and Leymann, F. (2008). **‘RESTful Web Services vs “Big” Web Services: Making the Right Architectural Decision’**, Proc. WWW 2008, Beijing, China: 805–14.
13. Richardson, L. and Ruby, S. (2007). **‘RESTful Web Services’**, Sebastopol, Calif.: O’Reilly Media Inc.
14. Turner, M., Budgen, D. and Brereton, P. (2003). **‘Turning Software into a Service’**, IEEE Computer, 36 (10), 38–45.
15. White, S. A. (2004a). **‘An Introduction to BPMN’**,
<http://www.bpmn.org/Documents/Introduction%20to%20BPMN>.
16. White, S. A. (2004b). **‘Process Modelling Notations and Workflow Patterns’**, In Workflow Handbook 2004. Fischer, L. (ed.). Lighthouse Point, Fla.: Future Strategies Inc. 265–294.
17. White, S. A. and Miers, D. (2008). BPMN Modeling and Reference Guide: Understanding and Using BPMN. Lighthouse Point, Fla.: Future Strategies Inc.

18.6 Exercises

- 18.1. What are the most important distinctions between services and software components?
- 18.2. Explain why SOAs should be based on standards.
- 18.3. Define an interface specification for the Currency Converter and Check credit rating services.
- 18.4. Design possible input and output messages for the services and specify these in the UML or in XML.
- 18.5. Giving reasons for your answer, suggest two important types of applications where you would not recommend the use of service-oriented architecture.
- 18.6. Using BPMN, design a workflow that uses the catalog service to look up and place orders for computer equipment.

- 18.7. Explain what is meant by a ‘compensation action’ and, using an example, show why these actions may have to be included in workflows.
- 18.8. For the example of the vacation package reservation service, design a workflow that will book ground transportation for a group of passengers arriving at an airport. They should be given the option of booking either a taxi or renting a car. You may assume that the taxi and car rental companies offer web services to make a reservation.
- 18.9. Using an example, explain in detail why the thorough testing of services that include compensation actions is difficult.

SOFTWARE REUSE

Unit Structure

19.0 Objectives

19.1 Introduction

19.2 Contents

19.2.1 Overview of Software Reuse

19.2.2 The Reuse Landscape

19.2.3 Application Frameworks

19.2.4 Software Product Lines

19.2.5 COTS Product Reuse

19.2.5.1 COTS – Solution Systems

19.2.5.2 COTS – Integrated Systems

19.3 Key Points

19.4 Bibliography

19.5 Exercises

19.0 Objectives

The objectives of this subdivision to initiate the software reuse and to describe approaches to system development based on large-scale system reuse.

- To explain the benefits of software reuse and some reuse problems;
- To discuss several different ways to implement software reuse;
- To explain how reusable concepts can be represented as patterns or embedded in program generators;
- To discuss COTS reuse
- To describe the development of software product lines;

19.1 Introduction

In most engineering disciplines, systems are designed by composing existing components that have been used in other systems. Software engineering has been

more focused on original development but it is now recognized that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on systematic reuse. The goal of software reuse is to reduce the cost of software production by replacing creation with recycling.

19.2 Content

19.2.1 Overview of Software Reuse

- Reuse-based software engineering is a software engineering strategy where the development process is geared to reusing existing software.
- Although reuse was proposed as a development strategy more than 40 years ago (McIlroy, 1968), it is only since 2000 that ‘development with reuse’ has become the norm for new business systems.
- Reuse-based software engineering is an approach to development that tries to maximize the reuse of existing software.
- The software units that are reused may be of radically different sizes.

For example:

1. Application system reuse

The whole of an application system may be reused by incorporating it without changing into other systems or by configuring the application for different customers. Alternatively, application families that have a common architecture, but which are tailored for specific customers, may be developed.

2. Component reuse

Component of an application, ranging in size from subsystems to single objects, may be reused. For example, a pattern-matching system developed as part of a text-processing system may be reused in a database management system.

3. Object and function reuse

Software components that implement a single function, such as a mathematical function, or an object class may be reused. This form of reuse, based around standard libraries, has been common for the past 40 years. Many libraries of functions and classes are freely available. You reuse the classes and functions in these libraries by linking them with newly developed application code. In areas such as mathematical algorithms and graphics, where specialized expertise is needed to develop efficient objects and functions, this is a particularly effective approach.

Table 19.1. Benefits of Software Reuse

Benefit	Explanation
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
Reduced process risk	The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.

Table 19.2. Problems with Reuse

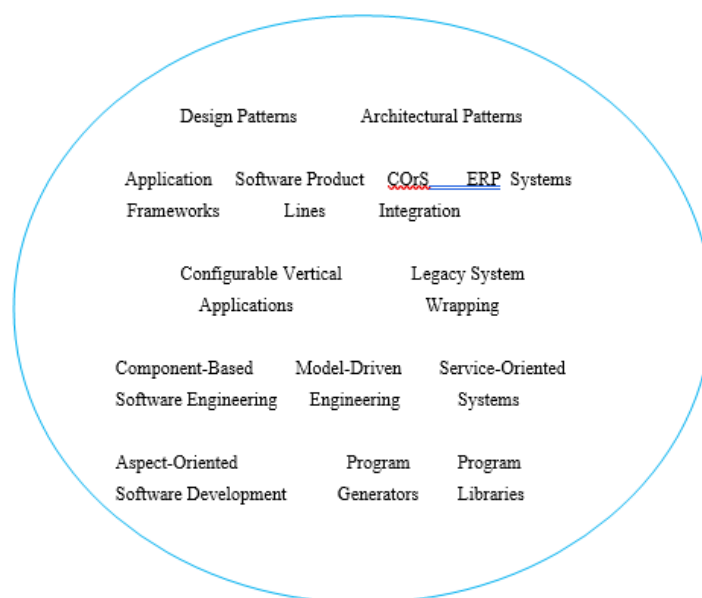
Problem	Explanation
Increased maintenance costs	If the source code of a reused software system or component is not available, then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is particularly true for tools that support embedded systems engineering, less so for object-oriented development tools.

Not-invented-here syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.

19.2.2 The Reuse Landscape

Over the past 20 years, many techniques have been developed to support software reuse. These techniques exploit the facts that systems in the same application domain are similar and have potential for reuse; that reuse is possible at different levels from simple functions to complete applications; and that standards for reusable components facilitate reuse. Figure 19.1, sets out a number of possible ways of implementing software reuse, with each described briefly in Table 19.3.

Figure 19.1. The Reuse Landscape



Key factors - Should consider when planning reuses are:

1. **The development schedule for the software:** If the software has to be developed quickly, you should try to reuse off-the-shelf systems rather than individual components. These are large-grain reusable assets.
2. **The expected software lifetime:** If you are developing a long-lifetime system, you should focus on the maintainability of the system. You should not just think about the immediate benefits of reuse but also of the long-term implications. Over its lifetime, you will have to adapt the system to new requirements, which will mean making changes to parts of the system. If you do not have access to the source code, you may prefer to avoid off-the-shelf components and systems from external suppliers; suppliers may not be able to continue support for the reused software.
3. **The background, skills, and experience of the development team:** All reuse technologies are fairly complex and you need quite a lot of time to understand and use them effectively. Therefore, if the development team has skills in a particular area, this is probably where you should focus.
4. **The criticality of the software and its non-functional requirements:** For a critical system that has to be certified by an external regulator, you may have to create a dependability case for the system. If your software has stringent performance requirements, it may be impossible to use strategies such as generator-based reuse, where you generate the code from a reusable domain-specific representation of a system. These systems often generate relatively inefficient code.
5. **The application domain:** In some application domains, such as manufacturing and medical information systems, there are several generic products that may be reused by configuring them to a local situation.
6. **The platform on which the system will run:** Some components models, such as .NET, are specific to Microsoft platforms. Similarly, generic application systems may be platform-specific and you may only be able to reuse these if your system is designed for the same platform.

Table 19.3. Approaches that support software reuse

Approach	Description
Architectural patterns	Standard software architectures that support common types of application systems are used as the basis of applications.
Design patterns	Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions.

Component-based development	Systems are developed by integrating components (collections of objects) that conform to component-model standards.
Application frameworks	Collections of abstract and concrete classes are adapted and extended to create application systems.
Legacy system wrapping	Legacy systems are ‘wrapped’ by defining a set of interfaces and providing access to these legacy systems through these interfaces.
Service-oriented systems	Systems are developed by linking shared services, which may be externally provided.
Software product lines	An application type is generalized around a common architecture so that it can be adapted for different customers.
COTS product reuse	Systems are developed by configuring and integrating existing application systems.
ERP systems	Large-scale systems that encapsulate generic business functionality and rules are configured for an organization.
Configurable vertical applications	Generic systems are designed so that they can be configured to the needs of specific system customers.
Program libraries	Class and function libraries that implement commonly used abstractions are available for reuse.
Model-driven engineering	Software is represented as domain models and implementation independent models and code is generated from these models.
Program generators	A generator system embeds knowledge of a type of application is used to generate systems in that domain from a user-supplied system model.
Aspect-oriented software development	Shared components are woven into an application at different places when the program is compiled.

19.2.3 Application Frameworks

A framework is a generic structure that is extended to create a more specific subsystem or application. Schmidt et al. (2004) define a frame-work to be:

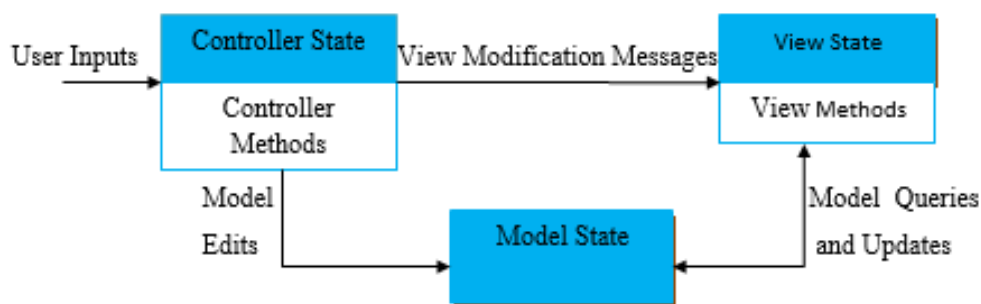
“An integrated set of software arte facts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications.”

Frameworks provide support for generic features that are likely to be used in all applications of a similar type. For example, a user interface framework will

provide support for interface event handling and will include a set of widgets that can be used to construct displays. It is then left to the developer to specialize these by adding specific functionality for a particular application. For example, in a user interface framework, the developer defines display layouts that are appropriate to the application being implemented.

Frameworks support design reuse in that they provide a skeleton architecture for the application as well as the reuse of specific classes in the system. The architecture is defined by the object classes and their interactions. Classes are reused directly and may be extended using features such as inheritance.

Figure 19.2. The Model-View-Controller patterns



Frameworks are implemented as a collection of concrete and abstract object classes in an object-oriented programming language. Therefore, frameworks are language-specific. There are frameworks available in all of the commonly used object-oriented programming languages (e.g., Java, C#, C++, as well as dynamic languages such as Ruby and Python). In fact, a framework can incorporate several other frameworks, where each of these is designed to support the development of part of the application. You can use a framework to create a complete application or to implement part of an application, such as the graphical user interface.

Fayad and Schmidt (1997) discuss three classes of frameworks:

1. **System infrastructure frameworks** - These frameworks support the development of system infrastructures such as communications, user interfaces, and compilers (Schmidt, 1997).
2. **Middleware integration frameworks** - These consist of a set of standards and associated object classes that support component communication and information exchange. Examples of this type of framework include Microsoft's .NET and Enterprise Java Beans (EJB). These frameworks provide support for standardized component models.

3. **Enterprise application frameworks** - These are concerned with specific application domains such as telecommunications or financial systems (Baumer, et al.,1997). These embed application domain knowledge and support the development of end-user applications.

Web Application Frameworks (WAFs)

Web Application Frameworks (WAFs) are a more recent and very important type of framework. WAFs that support the construction of dynamic websites are now widely available. The architecture of a WAF is usually based on the Model-View-Controller (MVC) composite pattern (Gamma et al., 1995), shown in Figure 19.2.

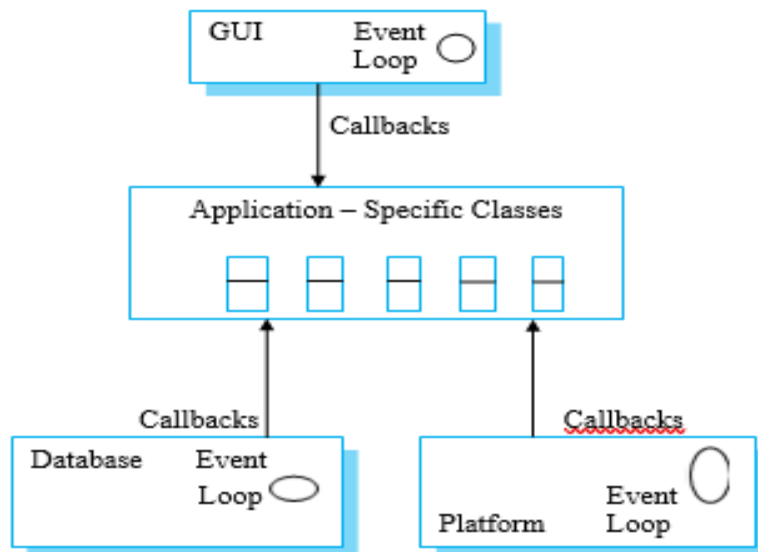
The MVC pattern was originally proposed in the 1980s as an approach to GUI design that allowed for multiple presentations of an object and separate styles of inter-action with each of these presentations. It allows for the separation of the application state from the user interface to application. An MVC framework supports the presentation of data in different ways and allows interaction with each of these presentations. When the data is modified through one of the presentations, the system model is changed and the controllers associated with each view update their presentation. For example, an MVC framework includes the Observer pattern, the Strategy pattern, the Composite pattern, and a number of others that are discussed by Gamma et al. (1995). The general nature of patterns and their use of abstract and concrete classes allows for extensibility. Without patterns, frameworks would, almost certainly, be impractical. Web application frameworks usually incorporate one or more specialized frame-works that support specific application features.

Although each framework includes slightly different functionality, most web application frameworks support the following features:

1. **Security** - WAFs may include classes to help implement user authentication (login) and access control to ensure that users can only access permitted functionality in the system.
2. **Dynamic web pages** - Classes are provided to help you define web page templates and to populate these dynamically with specific data from the system database.
3. **Database support** - Frameworks doesn't usually include a database but rather assume that a separate database, such as MySQL, will be used. The framework may provide classes that provide an abstract interface to different databases.
4. **Session management** - Classes to create and manage sessions (a number of inter-actions with the system by a user) are usually part of a WAF.

5. **User interaction** - Most web frameworks now provide AJAX support (Holdener, 2008), which allows more interactive web pages to be created.

Figure 19.3. Inversion of control in Frameworks



However, frameworks are usually more general than software product lines, which focus on a specific family of application system. For example, you can use a web-based framework to build different types of web-based applications. One of these might be a software product line that supports web-based help desks. This ‘help desk product line’ may then be further specialized to provide particular types of help desk support.

Frameworks are an effective approach to reuse, but are expensive to introduce into software development processes. They are inherently complex and it can take several months to learn to use them. It can be difficult and expensive to evaluate available frameworks to choose the most appropriate one. Debugging framework-based applications is difficult because you may not understand how the framework methods interact. This is a general problem with reusable software. Debugging tools may provide information about the reused system components, which a developer does not understand.

19.2.4 Software Product Lines

The most effective approaches to reuse are to create software product lines or application families. A software product line is a set of Applications with a common architecture and shared components, with each application specialized to reflect different requirements. The core system is designed to be configured and adapted to suit the needs of different system customers. This may involve the

configuration of some components, implementing additional components, and modifying some of the components to reflect new requirements.

Software product lines usually emerge from existing applications. That is, an organization develops an application then, when a similar system is required, informally reuses code from this in the new application. The same process is used as other similar applications are developed. However, change tends to corrupt application structure so, as more new instances are developed; it becomes increasingly difficult to create a new version. Consequently, a decision to design a generic product line may then be made. This involves identifying common functionality in product instances and including this in a base application, which is then used for future development.

Application frameworks and software product lines obviously have much in common. They both support a common architecture and components, and require new development to create a specific version of a system.

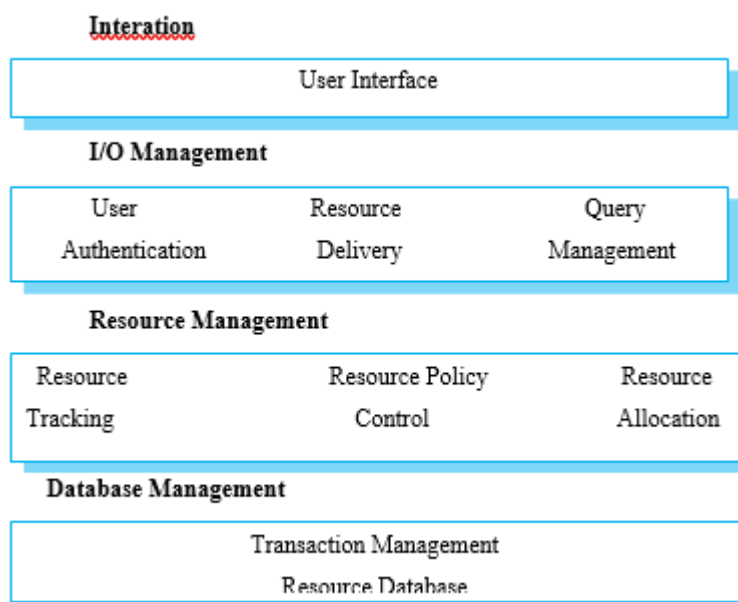
The main differences between these approaches are as follows:

1. Application frameworks rely on object-oriented features such as inheritance and polymorphism to implement extensions to the framework. Generally, the framework code is not modified and the possible modifications are limited to whatever is allowed by the framework. Software product lines are not necessarily created using an object-oriented approach. Application components are changed, deleted, or rewritten. There are no limits, in principle at least, to the changes that can be made.
2. Application frameworks are primarily focused on providing technical rather than domain-specific support. For example, there are application frameworks to create web-based applications. A software product line usually embeds detailed domain and platform information. For example, there could be a software product line concerned with web-based applications for health record management.
3. Software product lines are often control applications for equipment. For example, there may be a software product line for a family of printers. This means that the product line has to provide support for hardware interfacing. Application frameworks are usually software-oriented and they rarely provide support for hardware interfacing.
4. Software product lines are made up of a family of related applications, owned by the same organization. When you create a new application, your starting point is often the closest member of the application family, not the generic core application.

Various types of specialization of a software product line may be developed:

1. **Platform specialization** - Versions of the application are developed for different platforms. For example, versions of the application may exist for Windows, Mac OS, and Linux platforms. In this case, the functionality of the application is normally unchanged; only those components that interface with the hardware and operating system are modified.
2. **Environment specialization** - Versions of the application are created to handle particular operating environments and peripheral devices. For example, a system for the emergency services may exist in different versions, depending on the vehicle communications system. In this case, the system components are changed to reflect the functionality of the communications equipment used.
3. **Functional specialization** - Versions of the application are created for specific customers who have different requirements. For example, a library automation system may be modified depending on whether it is used in a public library, a reference library, or a university library. In this case, components that implement functionality may be modified and new components added to the system.
4. **Process specialization** - The system is adapted to cope with specific business processes. For example, an ordering system may be adapted to cope with a centralized ordering process in one company and a distributed process in another.

Figure 19.4. The Architecture of a Resource Allocation System

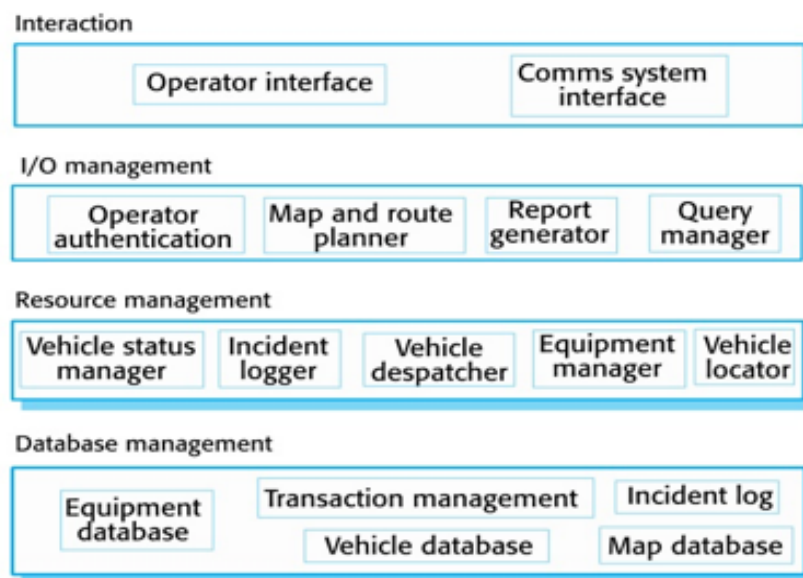


The architecture of a software product line often reflects a general, application specific architectural style or pattern. For example, consider a product line system that is designed to handle vehicle dispatching for emergency services. Operators of this system take calls about incidents, find the appropriate vehicle to respond to the incident and dispatch the vehicle to the incident site. The developers of such a system may market versions of this for police, fire, and ambulance services.

This vehicle dispatching system is an example of resource management architecture (Figure 19.4). You can see how this four-layer structure is instantiated in Figure 19.5, which shows the modules that might be included in a vehicle dispatching system product line. The components at each level in the product line system are as follows:

1. At the interaction level, there are components providing an operator display interface and an interface with the communications systems used.
2. At the I/O management level (level 2), there are components that handle operator authentication, generate reports of incidents and vehicles dispatched, support map output and route planning, and provide a mechanism for operators to query the system databases.
3. At the resource management level (level 3) there are components that allow vehicles to be located and dispatched, components to update the status of vehicles and equipment, and a component to log details of incidents.
4. At the database level, as well as the usual transaction management support, there are separate databases of vehicles, equipment, and maps.

Figure 19.5. The Product Line Architecture of a vehicle Dispatcher System

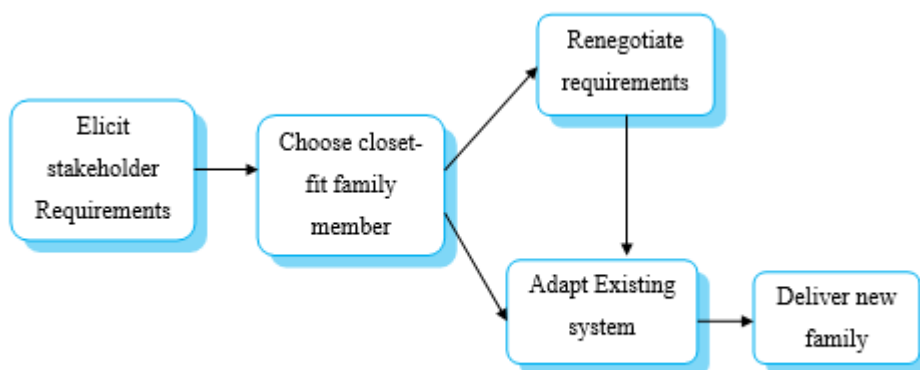


To create a specific version of this system, you may have to modify individual components. For example, the police have a large number of vehicles but a small number of vehicle types, whereas the fire service has many types of specialized vehicles. Therefore, you may have to define a different vehicle database structure when implementing a system for these different services.

Figure 19.6, shows the steps involved in extending a software product line to create a new application. The steps involved in this general process are as follows:

1. **Elicit stakeholder requirements** - You may start with a normal requirement engineering process. However, because a system already exists, you will need to demonstrate the system and have stakeholders' experiment with it, expressing their requirements as modifications to the functions provided.
2. **Select the existing system** - That is the closest fit to the requirements. When creating a new member of a product line, you may start with the nearest product instance. The requirements are analyzed and the family member that is the closest fit is chosen for modification.
3. **Renegotiate requirements** - As more details of required changes emerge and the project is planned, there may be some requirements renegotiation to minimize the changes that are needed.
4. **Adapt existing system** - new modules are developed for the existing system and existing system modules are adapted to meet the new requirements.
5. **Deliver new family member** - The new instance of the product line is delivered to the customer. At this stage, you should document its key features so that it may be used as a basis for other system developments in the future.

Figure 19.6. Product Instance Development



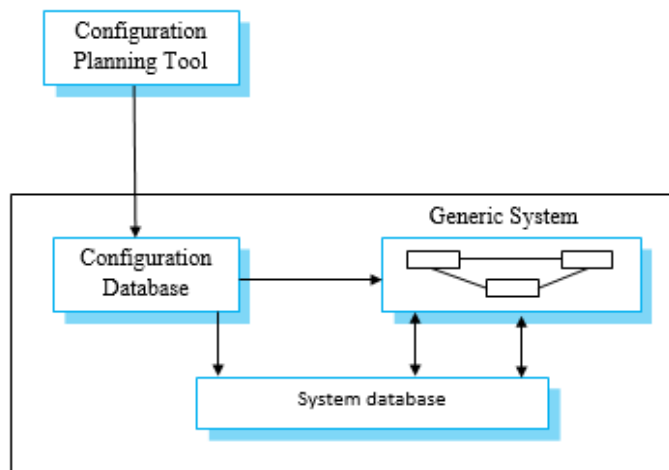
When you create a new member of product line you may have to find a compromise between reusing as much of the generic application as possible and satisfying detailed stakeholder requirements. The more detailed the system requirements, the less likely it is that the existing components will meet these requirements.

However, if stakeholders are willing to be flexible and to limit the system modifications that are required, you can usually deliver the system more quickly and at a lower cost. Software product lines are designed to be reconfigured and this reconfiguration may involve adding or removing components from the system, defining parameters and constraints for system components, and including knowledge of business processes.

This configuration may occur at different stages in the development process:

1. **Design-time configuration** - The organization that is developing the software modifies a common product line core by developing, selecting, or adapting components to create a new system for a customer.
2. **Deployment-time configuration** - A generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer's specific requirements and the system's operating environment is embedded in a set of configuration files that are used by the generic system.

Figure 19.7. Deployment-Time Configuration



Deployment-time configuration involves using a configuration tool to create a specific system configuration that is recorded in a configuration database or as a set of configuration files (Figure 19.7). The executing system consults this database when executing so that its functionality may be specialized to its execution context.

There are several levels of deployment-time configuration that may be provided in a system:

1. Component selection, where you select the modules in a system that provide the required functionality. For example, in a patient information system, you may select an image management component that allows you to link medical images (x-rays, CT scans, etc.) to the patient's medical record.
2. Workflow and rule definition, where you define workflows (how information is processed, stage by stage) and validation rules that should apply to information entered by users or generated by the system.
3. Parameter definition, where you specify the values of specific system parameters that reflect the instance of the application that you are creating. For example, you may specify the maximum length of fields for data input by a user or the characteristics of hardware attached to the system.

Deployment-time configuration can be very complex and it may take many months to configure the system for a customer. Large configurable systems may support the configuration process by providing software tools, such as a configuration planning tools, to support the configuration process.

19.2.5 COTS PRODUCT REUSE

A commercial-off-the-shelf (COTS) product is a software system that can be adapted to the needs of different customers without changing the source code of the system.

Torchiano and Morisio (2004) also discovered that using open-source products were often used as COTS products. That is, the open-source systems were used without change and without looking at the source code. COTS products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs.

Significant Benefits:

This approach to software reuse has been very widely adopted by large companies over the last 15 or so years, as it offers significant benefits over customized software development:

1. As with other types of reuses, more rapid deployment of a reliable system may be possible.
2. It is possible to see what functionality is provided by the applications and so it is easier to judge whether or not they are likely to be suitable. Other

companies may already use the applications so experience of the systems is available.

3. Some development risks are avoided by using existing software.
4. Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.
5. As operating platforms evolve, technology updates may be simplified as these are the responsibility of the COTS product vendor rather than the customer.

Problems:

Of course, this approach to software engineering has its own problems:

1. Requirements usually have to be adapted to reflect the functionality and mode of operation of the COTS product. This can lead to disruptive changes to existing business processes.
2. The COTS product may be based on assumptions that are practically impossible to change. The customer must therefore adapt their business to reflect these assumptions.
3. Choosing the right COTS system for an enterprise can be a difficult process especially as many COTS products are not well documented. Making the wrong choice could be disastrous as it may be impossible to make the new system work as required.
4. There may be a lack of local expertise to support systems development. Consequently, the customer has to rely on the vendor and external consultants for development advice. This advice may be biased and geared to selling products and services, rather than meeting the real needs of the customer.
5. The COTS product vendor controls system support and evolution. They may go out of business, be taken over, or may make change that cause difficulties for customers.

Two Types of COTS Product Reuse

19.2.5.1 COTS-Solution Systems

COTS-solution systems consist of a generic application from a single vendor that is configured to customer requirements.

19.2.5.2 COTS-Integrated Systems

COTS-integrated systems involve integrating two or more COTS systems (perhaps from different vendors) to create an application system. Table 19.4, summarizes the differences between these different approaches.

Table 19.4. COTS-Solution and COTS-Integrated systems

COTS-Solution Systems	COTS-Integrated Systems
Single product that provides the functionality required by a customer	Several heterogeneous system products are integrated to provide customized functionality
Based around a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

i) COTS – Solution Systems

COTS-solution systems are generic application systems that may be designed to support a particular business type, business activity, or sometimes, a complete business enterprise. For example, a COTS-solution system may be produced for dentists that handle appointments, dental records, patient recall, etc. At a larger scale, an Enterprise Resource Planning (ERP) system may support all of the manufacturing, ordering, and customer relationship management activities in a large company.

Domain-specific COTS-solution systems, such as systems to support a business function (e.g., document management), provide functionality that is likely to be required by a range of potential users. However, they also incorporate built-in assumptions about how users work and these may cause problems in specific situations.

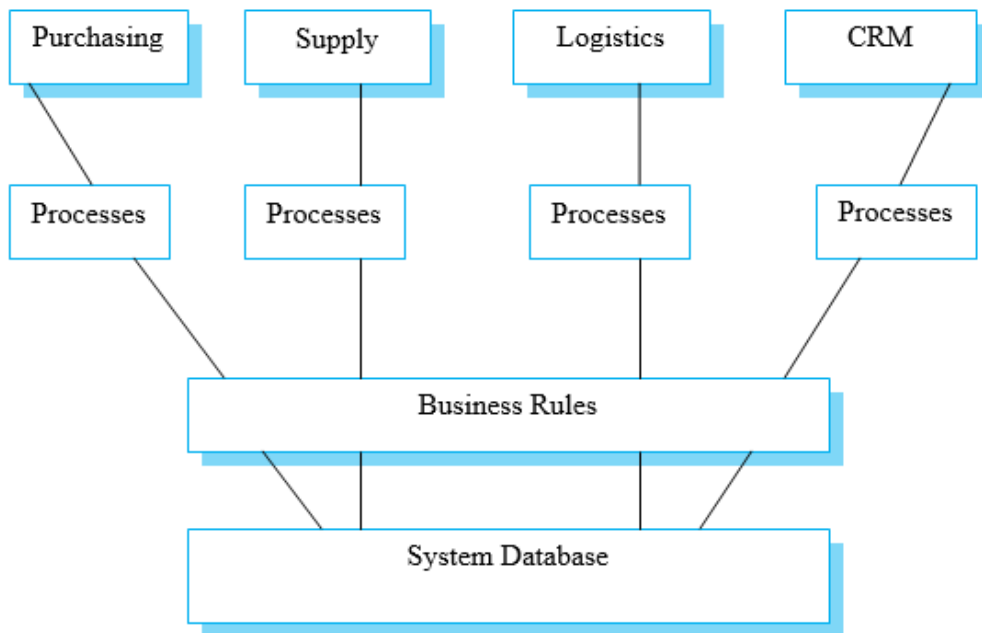
For example, a system to support student registration in a university may assume that students will be registered for one degree at one university. However, if universities collaborate to offer joint degrees, then it may be practically impossible to represent this in the system. ERP systems, such as those produced by SAP and BEA, are large-scale integrated systems designed to support business practices such as ordering and invoicing, inventory management, and manufacturing scheduling (O’Leary, 2000).

The configuration process for these systems involves gathering detailed information

about the customer's business and business processes, and embedding this in a configuration database. This often requires detailed knowledge of configuration notations and tools and is usually carried out by consultants working alongside system customers. A generic ERP system includes a number of modules that may be composed in different ways to create a system for a customer.

The configuration process involves choosing which modules are to be included, configuring these individual modules, defining business processes and business rules, and defining the structure and organization of the system database. A model of the overall architecture of an ERP system that supports a range of business functions is shown in Figure 19.8.

Figure 19.8. The Architecture of an ERP System



The key features of this architecture are:

1. A number of modules to support different business functions. These are large grain modules that may support entire departments or divisions of the business. In this example shown in Figure 19.8, the modules that have been selected for inclusion in the system are a module to support purchasing, a module to support supply chain management, a logistics module to support the delivery of goods, and a customer relationship management (CRM) module to maintain customer information.
2. A defined set of business processes, associated with each module, which relate to activities in that module. For example, there may be a definition of

the ordering process that defines how orders are created and approved. This will specify the roles and activities involved in placing an order.

3. A common database that maintains information about all related business functions. This means that it should not be necessary to replicate information, such as customer details, in different parts of the business.
4. A set of business rules that apply to all data in the database. Therefore, when data is input from one function, these rules should ensure that it is consistent with the data required by other functions. For example, there may be a business rule that all expense claims have to be approved by someone more senior than the person making the claim.

The configuration may involve:

1. Selecting the required functionality from the system (e.g., by deciding what modules should be included).
2. Establishing a data model that defines how the organization's data will be structured in the system database.
3. Defining business rules that apply to that data.
4. Defining the expected interactions with external systems.
5. Designing the input forms and the output reports generated by the system.
6. Designing new business processes that conform to the underlying process model supported by the system.
7. Setting parameters that define how the system is deployed on its underlying platform.

ii) COTS – Integrated Systems

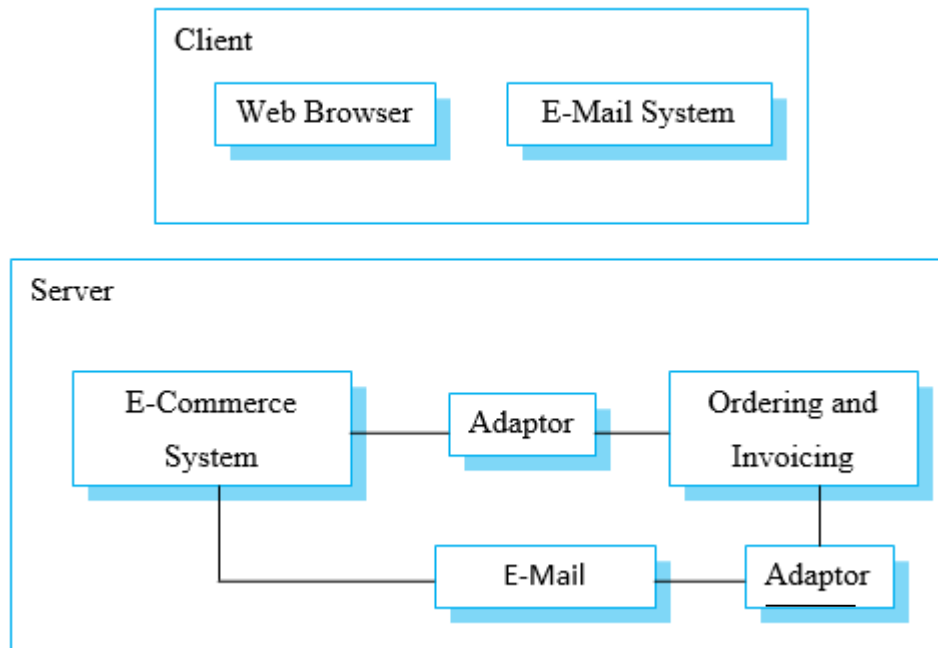
COTS-integrated systems are applications that include two or more COTS products or, sometimes, legacy application systems. You may use this approach when there is no single COTS system that meets all of your needs or when you wish to integrate a new COTS product with systems that you already use. The COTS products may interact through their APIs (Application Programming Interfaces) or service interfaces if these are defined.

Design choices:

1. Which COTS products offer the most appropriate functionality? Typically, there will be several COTS products available, which can be combined in different ways.
2. How will data be exchanged? Different products normally use unique data structures and formats.

- What features of a product will actually be used? COTS products may include more functionality than you need and functionality may be duplicated across different products.

Figure 19.9 A COTS-Integrated Procurement System



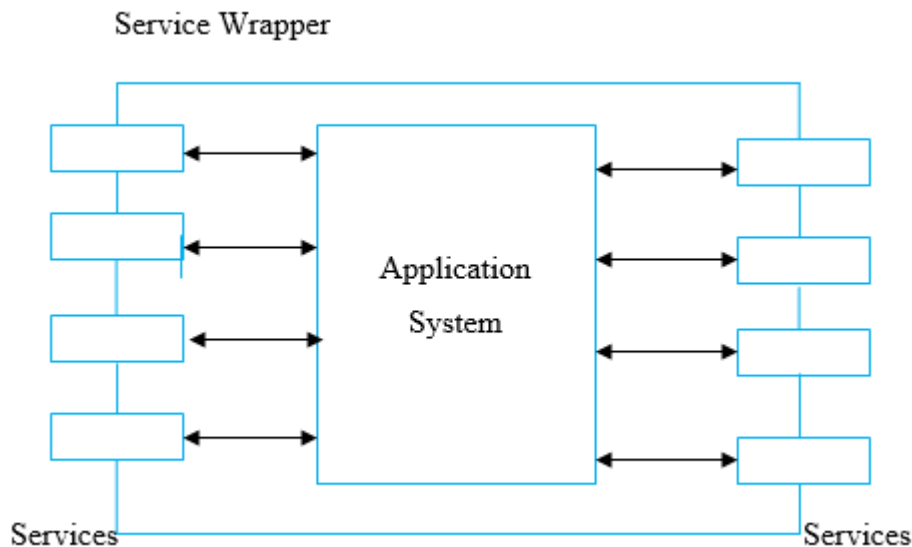
The structure of the final procurement system, constructed using COTS, is shown in Figure 19.9. This procurement system is a client–server based and, on the client, standard web browsing and e-mail software are used. On the server, the e-commerce platform has to integrate with the existing ordering system through an adaptor. The e-commerce system has its own format for orders, confirmations of delivery, and so forth, and these have to be converted into the format used by the ordering system. The e-commerce system uses the e-mail system to send notifications to users, but the ordering system was never designed for this. Therefore, another adaptor has to be written to convert the notifications from the ordering system into e-mail messages.

COTS integration can be simplified if a service-oriented approach is used. Essentially, a service-oriented approach means allowing access to the application system’s functionality through a standard service interface, with a service for each discrete unit of functionality. Some applications may offer a service interface but, sometimes, this service interface has to be implemented by the system integrator.

Essentially, you have to program a wrapper that hides the application and provides externally visible services (Figure 19.10). This approach is particularly valuable for legacy systems that have to be integrated with newer application systems.

In principle, integrating COTS products is the same as integrating any other components.

Figure 19.10. Application Wrapping



Four important COTS system integration problems:

1. **Lack of control over functionality and performance** - Although the published interface of a product may appear to offer the required facilities, these may not be properly implemented or may perform poorly. The product may have hidden operations that interfere with its use in a specific situation
2. **Problems with COTS system interoperability** - It is sometimes difficult to get COTS products to work together because each product embeds its own assumptions about how it will be used.
3. **No control over system evolution** - Vendors of COTS products makes their own decisions on system changes, in response to market pressures. For PC products, in particular, new versions are often produced frequently and may not be compatible with all previous versions.
4. **Support from COTS vendors** - The level of support available from COTS vendors varies widely. Vendor support is particularly important when problems arise as developers do not have access to the source code and detailed documentation of the system.

19.3 Key Points

- Most new business software systems are now developed by reusing knowledge and code from previously implemented systems.
- There are many different ways to reuse software. These range from the reuse of classes and methods in libraries to the reuse of complete application systems.
- The advantages of software reuse are lower costs, faster software development, and lower risks.
- System dependability is increased. Specialists can be used more effectively by concentrating their expertise on the design of reusable components.
- Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialization and the addition of new objects.
- They usually incorporate good design practice through design patterns. Software product lines are related applications that are developed from one or more base applications.
- A generic system is adapted and specialized to meet specific requirements for functionality, target platform, or operational configuration.
- COTS product reuse is concerned with the reuse of large-scale, off-the-shelf systems.
- These provide a lot of functionality and their reuse can radically reduce costs and development time.
- Systems may be developed by configuring a single, generic COTS product or by integrating two or more COTS products.
- Enterprise Resource Planning systems are examples of large-scale COTS reuse. You create an instance of an ERP system by configuring a generic system with information about the customer's business processes and rules.
- Potential problems with COTS-based reuse include lack of control over functionality and performance, lack of control over system evolution, the need for support from external vendors, and difficulties in ensuring that systems can interoperate.

19.4 Bibliography

1. **‘Software Engineering’**, Ian Somerville, Pearson Education. Ninth Edition.
2. **‘Software Engineering’**, Roger Pressman, Tata-Mcgraw Hill, Seventh Edition.
3. Baker, T. (2002). **‘Lessons Learned Integrating COTS into Systems’**. Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems), Orlando, Fla.: Springer, 21–30.
4. Balk, L. D. and Kedia, A. (2000). **‘PPT: A COTS Integration Case Study’**. Proc. Int. Conf. on Software Eng., Limerick, Ireland: ACM Press, 42–9.
5. Baumer,D., Gryczan,G., Knoll,R., Lilienthal,C., Riehle,D. and Zullighoven,H. (1997). **‘Framework Development for Large Systems’**. Comm.ACM, 40(10),52– 9.
6. Boehm, B. and Abts, C. (1999). **‘COTS Integration: Plug and Pray?’** IEEE Computer, 32 (1), 135–38.
7. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). **‘Design Patterns: Elements of Reusable Object-Oriented Software. Reading’**, Mass.: Addison Wesley.
8. Jacobson, I., Griss, M. and Jonsson, P. (1997). **‘Software Reuse. Reading, Mass’**, Addison-Wesley.

19.5 Exercises

- 19.1. What are the major technical and nontechnical factors that hinder software reuse? Do you personally reuse much software and, if not, why not?
- 19.2. Suggest why the savings in cost from reusing existing software are not simply proportional to the size of the components that are reused.
- 19.3. Give four circumstances where you might recommend against software reuse.
- 19.4. Explain what is meant by ‘inversion of control’ in application frameworks. Explain why this approach could cause problems if you integrated two separate systems that were originally created using the same application framework.

- 19.5. Most desktop software, such as word processing software, can be configured in a number of different ways. Examine software that you regularly use and list the configuration options for that software. Suggest difficulties that users might have in configuring the software. If you use Microsoft Office or Open Office, these are good examples to use for this exercise.
- 19.6. Why have many large companies chosen ERP systems as the basis for their organizational information system? What problems may arise when deploying a large-scale ERP system in an organization?
- 19.7. Identify six possible risks that can arise when systems are constructed using COTS. What steps can a company take to reduce these risks?
- 19.8. Explain why adaptors are usually needed when systems are constructed by integrating COTS products. Suggest three practical problems that might arise in writing adaptor software to link two COTS application products.
- 19.9. The reuse of software raises a number of copyright and intellectual property issues. If a customer pays a software contractor to develop a system, who has the right to reuse the developed code? Does the software contractor have the right to use that code as a basis for a generic component? What payment mechanisms might be used to reimburse providers of reusable components? Discuss these issues and other ethical issues associated with the reuse of software.

DISTRIBUTED SOFTWARE ENGINEERING

Unit Structure

- 20.1 Objectives
- 20.2 Introduction
- 20.3 Contents
 - 20.3.1 Overview of Distributed Software Engineering
 - 20.3.2 Distributed System Issues
 - 20.3.2.1 Models of Interaction
 - 20.3.2.2 Middleware
- 20.4 Client-Server Computing
- 20.5 Architectural patterns for Distributed Systems
 - i. Master-Slave Architectures
 - ii. Two-Tier Client-Server Architectures
 - iii. Multi-Tier Client-Server Architectures
 - iv. Distributed Component Architectures
 - v. Peer-to-Peer Architectures
- 20.6 Software as a Service (SaaS)
- 20.7 Key Points
- 20.8 Bibliography
- 20.9 Exercises

20.0 Objectives

Distributed software engineering is therefore very important for enterprise computing systems. The objective of this chapter is to introduce distributed systems engineering and distributed systems architectures.

When you have studied this chapter, you will:

- Know the key issues that have to be considered when designing and implementing distributed software systems;
- Understand the client-server computing model and the layered architecture of client-server systems;
- Have been introduced to commonly used patterns for distributed systems architectures and know the types of system for which each architecture is most applicable;
- Understand the notion of software as a service, providing web-based access to remotely deployed application systems.

20.2 Introduction

The term "Distributed Software Engineering" is ambiguous. It includes both the engineering of distributed software and the process of distributed development of software, such as cooperative work. It concentrates on the former, giving an indication of the special needs and rewards in distributed computing. It is an area of software engineering that is concerned with the development and the maintenance of reliable software for distributed systems.

20.3 Contents

20.3.1 Overview of Distributed Software Engineering

- Virtually all large computer-based systems are now distributed systems.
- Information processing is distributed over several computers rather than confined to a single machine.

Tanenbaum and Van Steen (2007) define a distributed system to be:

“A collection of independent computers that appears to the user as a single coherent system.”

Distributed System Characteristics/Advantages

- **Resource Sharing** – Sharing of Hardware and Software resources.
- **Openness** – Standard protocols allow equipment and software from different vendors to be combined.
- **Concurrency** – Parallel processing to enhance performance.
- **Scalability** – Increased throughput by adding new resources up to capacity of network.
- **Fault Tolerance** – Potential to continue in operation after a fault has occurred.

Distributed System Disadvantages

- **Complexity** – Typically, distributed systems are more complex than centralized systems.
- **Security** – More susceptible to external attack.
- **Manageability** – More effort required for system management.
- **Unpredictability** - Unpredictable responses depending on the system organization and network load.

20.3.2 Distributed System Issues

Distributed systems are more complex than systems that run on a single processor. This complexity arises because it is practically impossible to have a top-down model of control of these systems.

Design Issues

- **Transparency** - To what extent should the distributed system appear to the user as a single system? When it is useful for users to understand that the system is distributed?

Ideally, users should not be aware that a system is distributed and services should be independent of how they are distributed. In practice, this is impossible because parts of the system are independently managed and because of network delays. To achieve Transparency, resources should be abstracted and addressed logically rather than physically. Middleware maps logical to physical resources.

- **Openness** - Should a system be designed using standard protocols that support interoperability or should more specialized protocols be used that restrict the freedom of the designer?

Open distributed systems are systems that are built according to generally accepted standards. Components from any supplier developed in any programming language can be integrated into the system and can inter-operate with one another. Web service standards for service-oriented architectures were developed to be open standards.

- **Scalability** - How can the system be constructed so that it is scalable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?

The ability of a system to deliver a high-quality service as demands on the system increase :

Size – By adding more resources to cope with an increasing number of users.

Distribution – By geographically dispersing components without degrading performance.

Manageability – By effectively managing a system as it increases in size, even if its components are located in independent organizations.

- **Security** - How can usable security policies be defined and implemented that applies across a set of independently managed systems?

The number of ways that distributed systems may be attacked is significantly greater than for centralized systems. If a part of the system is compromised then the attacker may be able to use this as a “back door” into other parts of the system. Difficulties can arise when different organizations own parts of the system.

- **Quality of service** - How should the quality of service that is delivered to system users be specified and how should the system be implemented to deliver an acceptable quality of service to all users?

Reflects a system's ability to deliver services dependably and with a response time and throughput that is acceptable to users. QoS is particularly critical when a system deals with time-critical data such as audio or video streams.

- **Failure management** - How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?

Since failures are inevitable, distributed systems must be designed to be resilient. Distributed systems should include mechanisms for discovering failed components, continuing to deliver as many services as possible and where possible, automatically from failures.

20.3.2.1 Models of Interaction

There are two fundamental types of interaction between the computers in a distributed computing system:

- **Procedural interaction** – Where one computer calls on a known service offered by another computer and waits for a response.(synchronous)
- **Message based interaction** - Involves the 'sending' computer defining information about what is required in a message, which is then sent to another computer. There is no necessity to wait for a response. (non-synchronous)

Figure 20.1. Procedural interaction between a Dinner and a Waiter via Synchronous procedure call

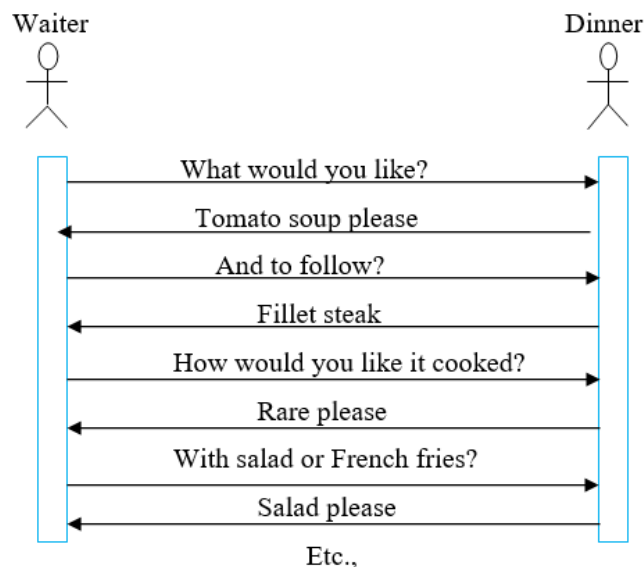


Figure 20.2. Message-Based Interaction between a Waiter and the Kitchen Staff

```

<starter>
<dish name = "soup" type = "tomato" />
<dish name = "soup" type = "fish" />
<dish name = "pigeon salad" />
</starter>
<main course>
<dish name = "steak" type = "sirloin" cooking = "medium" />
<dish name = "steak" type = "fillet" cooking = "rare" />
<dish name = "sea bass">
</main>
<accompaniment>
<dish name="french fries" portions="2"/> <dish name = "salad" portions =
"1"/>
</accompaniment>

```

Procedural (Synchronous) Interaction

- Implemented by Remote Procedure Calls (RPC's).
- One component call another (via a "stub") as if it were a local procedure or method.
- Middleware intercepts the call and passes it to the remote component which carries out the required computation and returns the result.
- A problem is that both components need to be available at the time of the communication and must know how to refer to each other.

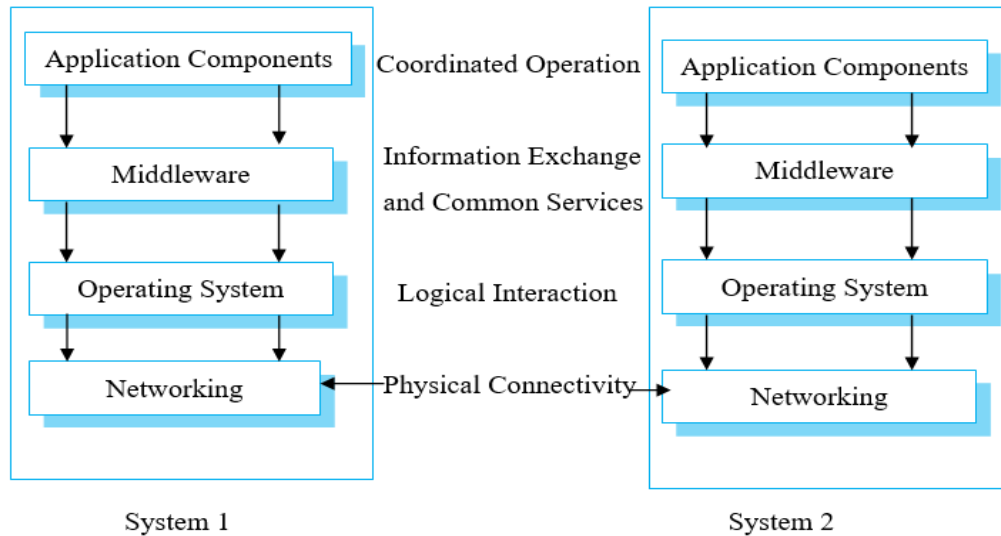
Message-based (Asynchronous) interaction

- Normally involves one component creating a message that details the services required of another.
- Message is sent via middleware to the receiving component which parses the message, carries out the computations, and (possibly) creates a return message with the required results.
- Messages are queued until the receiver is available components need NOT refer to each other; middleware ensures that messages are passed to the appropriate component.

20.3.2.2 Middleware

- The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor.

- Models of data, information representation, and protocols for communication may all be different. A distributed system therefore requires software that can manage these diverse parts, and ensure that they can communicate and exchange data.
- **Examples include CORBA, DCOM, .NET**



Two distinct types of Middleware support:

1. **Interaction support** - where the middleware coordinates interactions between different components in the system. The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components.
2. **Common services** - where the middleware provides reusable implementations of services that may be required by several components in the distributed system (i.e., Security, notification, transaction management etc.). By using these common services, components can easily interoperate and provide user services in a consistent way.

20.4 Client – Server Computing

- Distributed systems that are accessed over the internet are often organized as client-server(C/S) systems.
- The user interacts with a program running on a local computer (e.g., a web browser or phone-based application) which interacts with another program running on a remote computer (e.g., a web server).
- The remote computer provides services, such as access to web pages, which are available to clients.

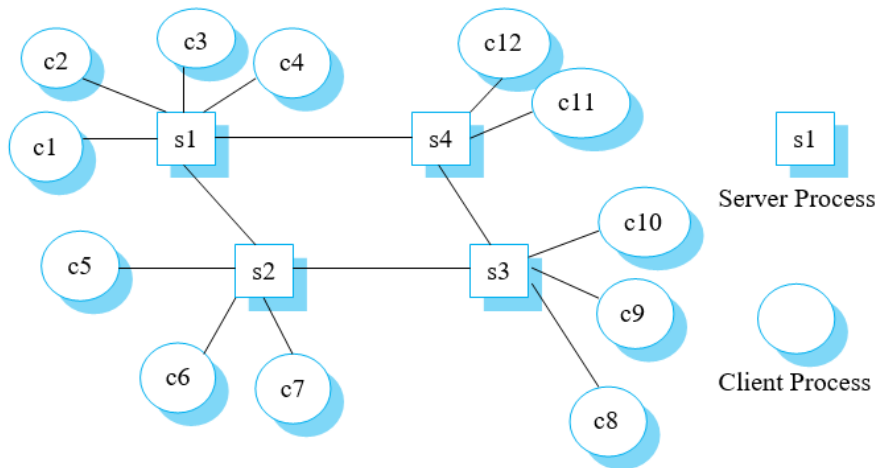
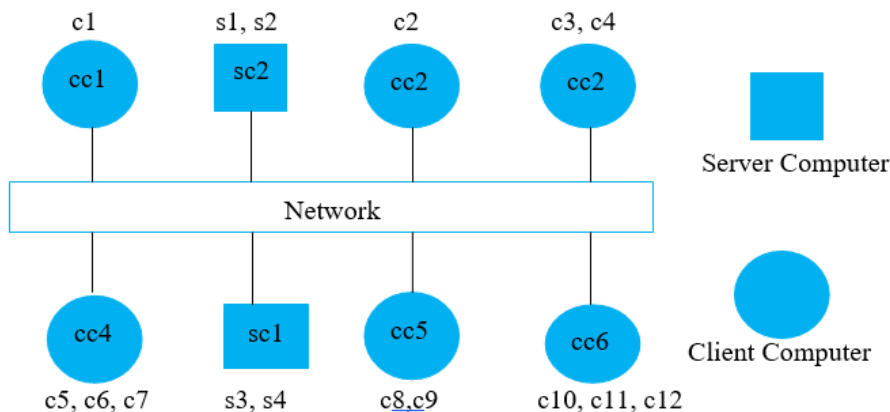
Figure 20.4. Processes in a Client-Server system

Figure 20.4, shows client and server processes rather than processors. It is normal for several client processes to run on a single processor. For example, on your PC, you may run a mail client that downloads mail from a remote mail server. You may also run a web browser that interacts with a remote web server and a print client that sends documents to a remote printer

Figure 20.5, illustrates the situation where the 12 logical clients shown in Figure 20.4, are running on six computers. The four server processes are mapped onto two physical server computers. Several different server processes may run on the same processor but, often, servers are implemented as multiprocessor systems in which a separate instance of the server process runs on each machine. Load-balancing software distributes requests for service from clients to different servers so that each server does the same amount of work. This allows a higher volume of transactions with clients to be handled, without degrading the response to individual clients.

Figure 20.5. Mapping of Client and Server processes to networked computers

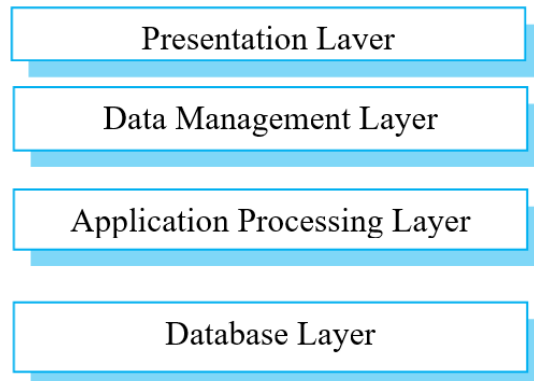
Client–server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information. Consequently, you should design the architecture of distributed client server systems so that they are structured into several logical layers, with clear interfaces between these layers. This allows each layer to be distributed to a different computer.

Four Layers

Figure 20.6 illustrates this model, showing an application structured into four layers:

- A presentation layer that is concerned with presenting information to the user and managing all user interaction;
- A data management layer that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, etc.;
- An application processing layer that is concerned with implementing the logic of the application and so providing the required functionality to end users;
- A database layer that stores the data and provides transaction management services, etc.

Figure 20.6. Layered architectural model for client–server application



20.5 Architectural Patterns For Distributed Systems

Five architectural styles:

1. **Master-slave architecture** - which is used in real-time systems in which guaranteed interaction response times are required.
2. **Two-tier client/server architecture** - which is used for simple client–server systems, and in situations where it is important to centralize the

system for security reasons. In such cases, communication between the client and server is normally encrypted.

3. **Multitier client–server architecture** - which is used when there is a high volume of transactions to be processed by the server.
4. **Distributed component architecture** - which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client/server systems.
5. **Peer-to-peer architecture** - which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

i. Master-Slave Architectures

Master-slave architectures for distributed systems are commonly used in real time systems where there may be separate processors associated with data acquisition from the system's environment, data processing, and computation and actuator management.

- The '**Master**' process is usually responsible for computation, coordination, and communications and it controls the 'slave' processes.
- '**Slave**' processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

Figure 20.7. A Traffic management system with a Master-Slave Architecture

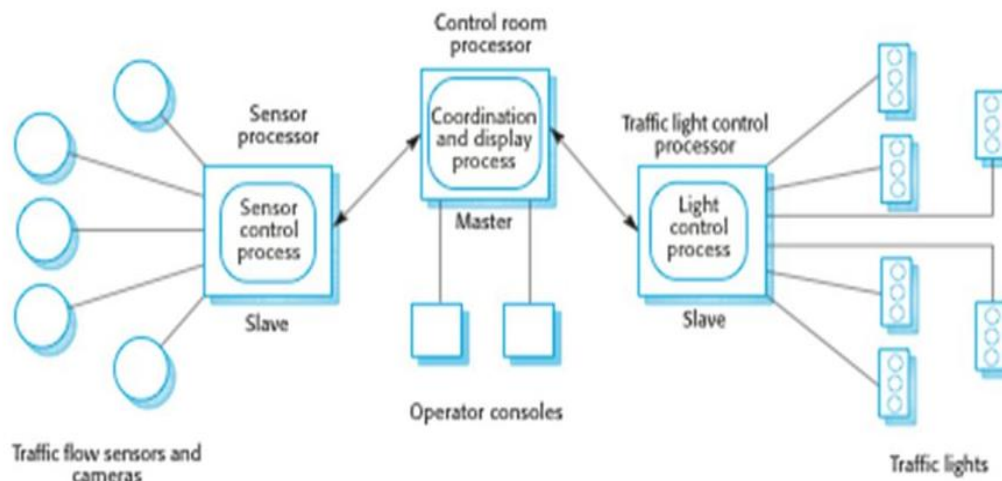


Figure 20.7, illustrates this architectural model. It is a model of a traffic control system in a city and has three logical processes that run on separate processors. The master process is the control room process, which

communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.

- A set of distributed sensors collects information on the traffic flow. The sensor control process polls the sensors periodically to capture the traffic flow information and collates this information for further processing.
- The sensor processor is itself polled periodically for information by the master process that is concerned with displaying traffic status to operators, computing traffic light sequences and accepting operator commands to modify these sequences.
- The control room system sends commands to a traffic light control process that converts these into signals to control the traffic light hardware.
- The master control room system is itself organized as a client–server system, with the client processes running on the operator’s consoles.

ii. Two-Tier Client-Server Architectures

Two-tier client–server architecture is the simplest form of client–server architecture. The system is implemented as a single logical server plus an indefinite number of clients that use that server.

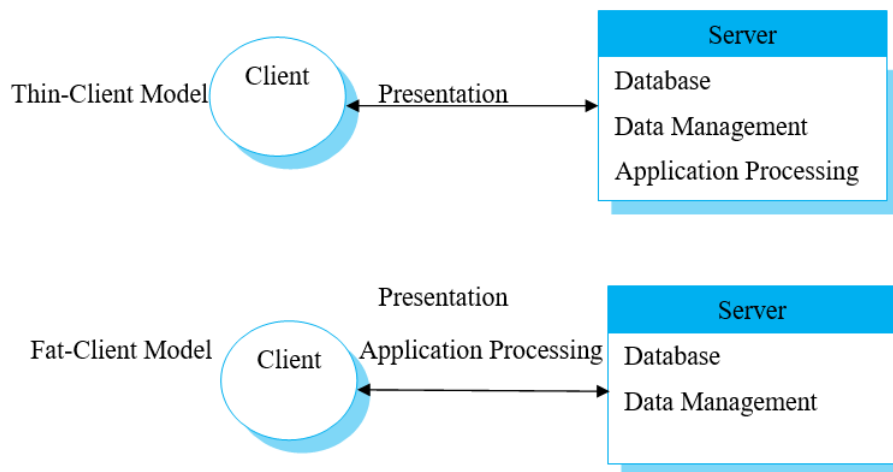
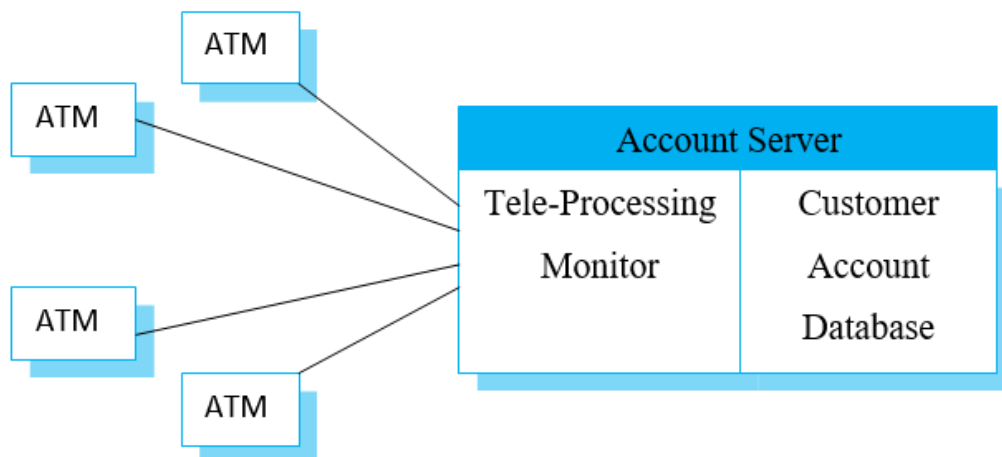
Two forms of Model

This is illustrated in Figure 20.8, which shows two forms of this architectural model:

1. **A Thin-Client model** - where the presentation layer is implemented on the client and all other layers (data management, application processing, and database) are implemented on a server. The client software may be a specially written program on the client to handle presentation. More often, however, a web browser on the client computer is used for presentation of the data.

Disadvantages : Places a heavy processing load on both the server and the network.

2. **A Fat-Client model** - where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server. Most suitable for new C/S systems where client capabilities are known in advance. System management is more complex when application functionality changes, updates are required on each client,

Figure 20.8. Thin- and Fat-Client Architectural Models**Figure 20.9. A Fat-client Architecture for an ATM system**

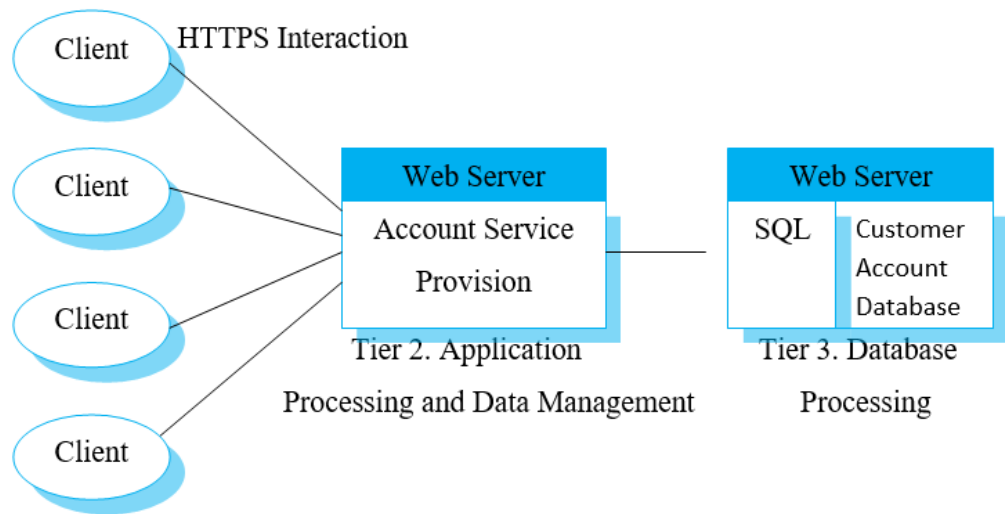
iii. Multi-Tier Client-Server Architectures

The fundamental problem with a two-tier client server approach is that the logical layers in the system presentation, application processing, data management, and database must be mapped onto two computer systems: the client and the server.

- Each layer may execute on a separate processor.
- Allows for :
 - Better Performance and scalability than the thin-client approach and is simpler to manage than a fat-client approach.

Figure 20.10. Three-Tier Architecture for an Internet Banking System

Tier 1. Presentation

**Table 20.1. Use of Client–Server Architectural patterns**

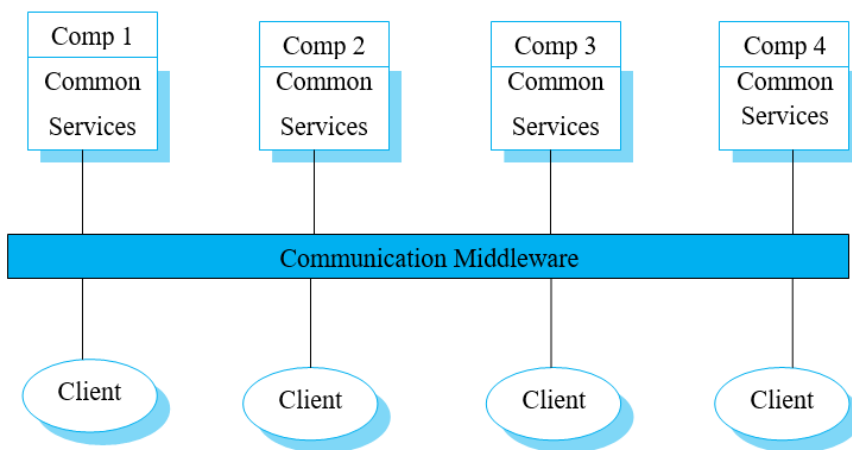
Architecture	Applications
Two-tier client–server architecture with thin clients	Legacy system applications that are used when separating application processing and data management is impractical. Clients may access these as services. Computationally intensive applications such as compilers with little or no data management. Data-intensive applications (browsing and querying) with non-intensive application processing. Browsing the Web is the most common example of a situation where this architecture is used.
Two-tier client-server architecture with fat clients	Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client. Applications where computationally intensive processing of data (e.g., data visualization) is required. Mobile applications where internet connectivity cannot be guaranteed. Some local processing using cached information from the database is therefore possible.

Multi-tier client–server architecture	Large-scale applications with hundreds or thousands of clients. Applications where both the data and the application are volatile. Applications where data from multiple sources are integrated.
---------------------------------------	---

iv. Distributed Component Architectures

System is designed as a set of services, without attempting to allocate these services to layers in the system. Each service, or group of related services, is implemented using a separate component. In a distributed component architecture (Figure 20.11) the system is organized as a set of interacting components or objects. Components communicate via Middleware using remote procedure or method calls.

Figure 20.11. A Distributed component Architecture



Distributed component systems are reliant on middleware, which manages component interactions, reconciles differences between types of the parameters passed between components, and provides a set of common services that application components can use. CORBA (Orfali et al., 1997) was an early example of such middle ware but is now not widely used. It has been largely supplanted by proprietary software such as Enterprise Java Beans (EJB) or .NET.

Advantages of Distributed Component Architectures

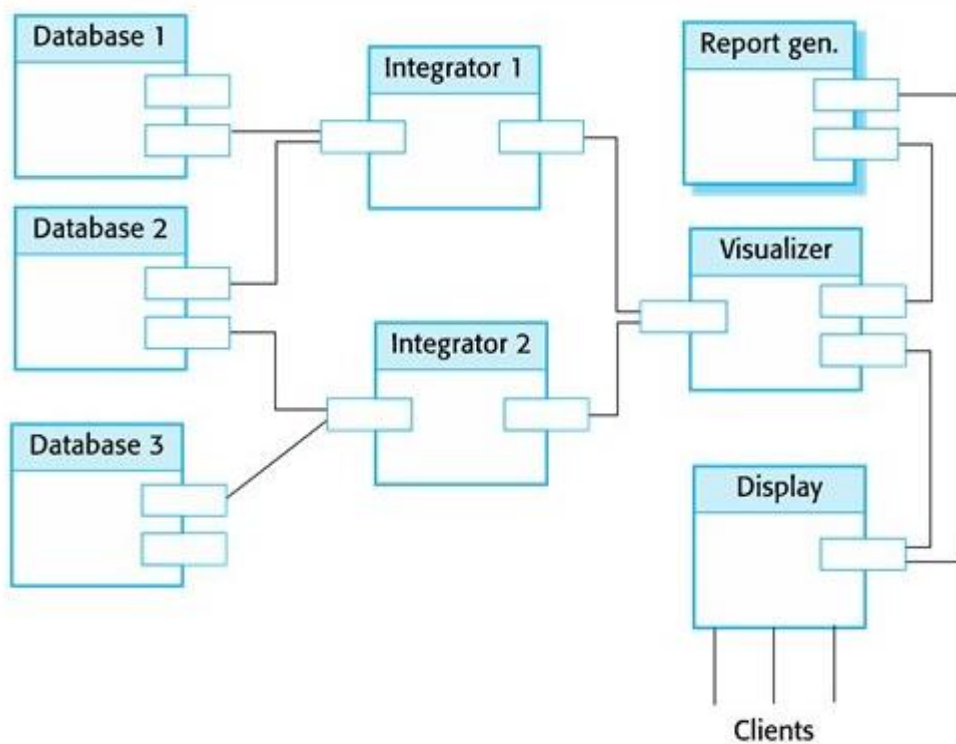
- Allows developers to delay decisions on where and how services should be provided.
- Very open Architecture – new resources can be added as required.

- System is dynamically reconfigurable – components can migrate across the network as required.
- **Therefore : Flexible and Scalable.**

Disadvantages of Distributed Component Architectures

- Less intuitive/natural than C/S – more difficult to visualize, understand and design.
- Competing middleware standards – vendors, such as Microsoft and Sun, have developed different incompatible middleware systems.

Figure 20.12. A Distributed component Architecture for a Data Mining System



Data mining systems are a good example of a type of system in which distributed component architecture is the best architectural pattern to use. A data mining system looks for relationships between the data that is stored in a number of data bases (Figure 20.12). Data mining systems usually pull in information from several separate databases, carry out computationally intensive processing, and display their results graphically.

v. Peer-to-Peer Architectures

Peer-to-peer (p2p) systems are decentralized systems in which computations may be carried out by any node on the network. In principle at least, no distinctions are

made between clients and servers. In peer-to-peer applications, the overall system is designed to take advantage of the computational power and storage available across a potentially huge network of computers. The standards and protocols that enable communications across the nodes are embedded in the application itself and each node must run a copy of that application.

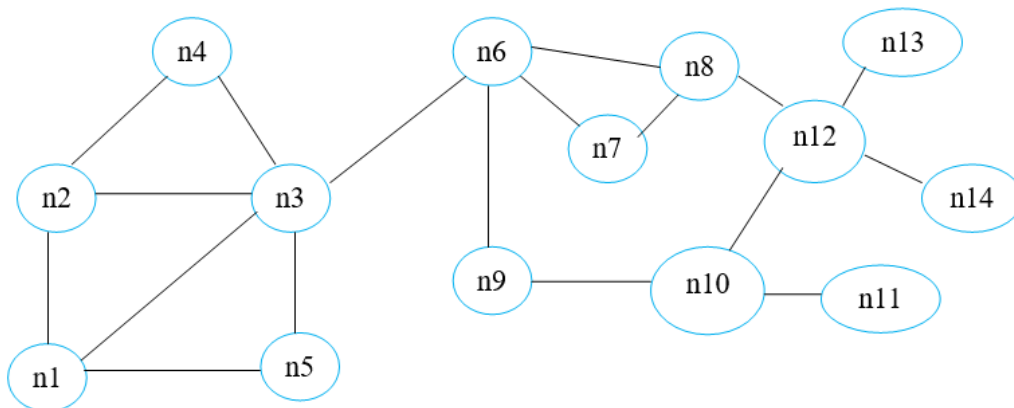
Peer-to-peer technologies have mostly been used for personal rather than business systems (Oram, 2001). For example, file-sharing systems based on the Gnutella and BitTorrent protocols are used to exchange files on users' PCs. However, peer-to-peer systems are also being used by businesses to harness the power in their PC networks (McDougall, 2000). Intel and Boeing have both implemented p2p systems for computationally intensive applications.

Appropriate uses of p2p

1. Computationally intensive applications where processing can be efficiently distributed among a large number of networked computers that need not communicate with one another.
2. Applications where processing involves a large number of networked computers exchanging data that need not be centrally stored or managed.

Decentralized p2p Architecture

Figure 20.13. A Decentralized p2p Architecture



In a decentralized architecture, the nodes in the network are not simply functional elements but are also communications switches that can route data and control signals from one node to another. For example, assume that Figure 20.13, represents a decentralized, document management system. This system is used by a consortium of researchers to share documents, and each member of the consortium maintains his or her own document store. However, when a document

is retrieved, the node retrieving that document also makes it available to other nodes.

If someone needs a document that is stored somewhere on the network, they issue a search command, which is sent to nodes in their 'locality'. These nodes check whether they have the document and, if so, return it to the requestor. If they do not have it, they route the search to other nodes. Therefore, if n1 issues a search for a document that is stored at n10, this search is routed through nodes n3, n6, and n9 to n10. When the document is finally discovered, the node holding the document then sends it to the requesting node directly by making a peer-to-peer connection.

Advantages

This decentralized architecture has advantages in that it is highly redundant and hence both fault-tolerant and tolerant of nodes disconnecting from the network.

Disadvantages

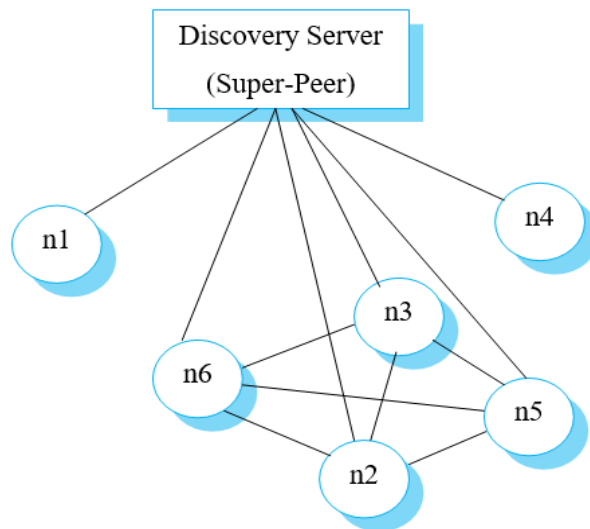
The disadvantages here are that many different nodes may process the same search, and there is also significant overhead in replicated peer communications.

Semicentralized p2p Architecture

An alternative p2p architectural model, which departs from pure p2p architecture, is a semicentralized architecture where, within the network, one or more nodes act as servers to facilitate node communications. This reduces the amount of traffic between nodes. Figure 20.14 illustrates this model. In a semicentralized architecture, the role of the server (sometimes called a super-peer) is to help establish contact between peers in the network, or to coordinate the results of a computation.

For example, if Figure 20.14, represents an instant messaging system, then network nodes communicate with the server (indicated by dashed lines) to find out what other nodes are available. Once these nodes are discovered, direct communications can be established and the connection to the server is unnecessary. Therefore, nodes n2, n3, n5, and n6 are in direct communication.

In a computational p2p system, where a processor-intensive computation is distributed across a large number of nodes, it is normal for some nodes to be super-peers. Their role is to distribute work to other nodes and to collate and check the results of the computation. Peer-to-peer architectures allow for the efficient use of capacity across a network.

Figure 20.14. A Semicentralized p2p Architecture**p2p Problems**

- The major concerns that have inhibited p2p use are **security** and **trust**.
- When p2p nodes interact with one another, any resources could potentially be accessed.
- Problems may also occur if peers on a network deliberately behave in a malicious way.

Example

There have been cases where music companies who believe that their copyright is being abused have deliberately made ‘poisoned peers’ available. When another peer downloads what they think is a piece of music, the actual file delivered is malware that may be a deliberately corrupted version of the music or a warning to the user of copyright infringement.

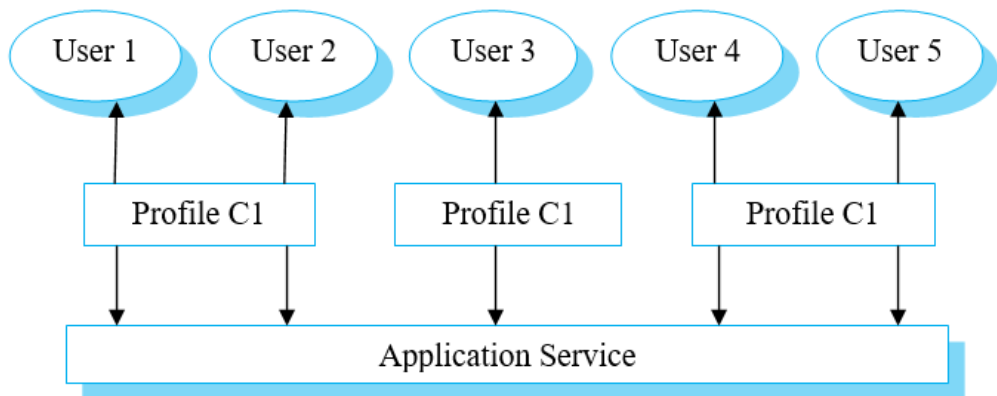
20.6 SOFTWARE as a SERVICE (SaaS)

- Involves hosting software remotely on servers (“the cloud”) with access provided over the Internet via web browsers.
- The server maintains user data and state during transaction session.
- Applications are owned and managed by a software provider rather than users.
- Users may pay for access according to the amount of use or through an annual or monthly subscription.
- If free, users are exposed to advertisements which fund the software service.

Difference between SaaS and Service Oriented Architecture (SOA)

1. SaaS is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions (e.g., editing document).
2. SOA is an approach to structuring a software system as a set of separate, stateless services. These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something, and then returns a result. Existing services may be composed and configured to create new composite services and applications. The basis for service composition is often a workflow.

Figure 20.15. Configuration of a Software System offered as a Service



Three factors into account

1. **Configurability** - How do you configure the software for the specific requirements of each organization?
2. **Multi-tenancy** - How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
3. **Scalability** - How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

Configuration

Configuration facilities may allow for the following:

1. **Branding** - where users from each organization, are presented with an interface that reflects their own organization.

2. **Business rules and workflows** - where each organization defines its own rules that govern the use of the service and its data.
3. **Database extensions** - where each organization defines how the generic service data model is extended to meet its specific needs.
4. **Access control**, - where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

Figure 20.15, illustrates this situation. This diagram shows five users of the application service, who work for three different customers of the service provider. Users interact with the service through a customer profile that defines the service configuration for their employer.

Multi-tenancy

Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources. However, it must appear to each user that they have the sole use of the system. Multi-tenancy involves designing the system so that there is an absolute separation between the system functionality and the system data.

The service provider can use a single database with different users being virtually isolated within that database. This is illustrated in Table 20.2, where you can see that database entries also have a ‘tenant identifier’, which links these entries to specific users. By using database views, you can extract the entries for each service customer and so present users from that customer with a virtual, personal database.

Table 20.2. A Multi-tenant Database

Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	Big Corp	2, Main st, Motown
435	X234	J. Bowie	56, Mill St, Starvile
592	PP37	R. Burns	Alloway, Ayrshire

Scalability

Scalability is the ability of the system to cope with increasing numbers of users without reducing the overall QoS that is delivered to any user. Generally, when

considering scalability in the context of SaaS, you are considering ‘scaling out’, rather than ‘scaling up’.

General guidelines for implementing scalable software are:

1. Develop applications where each component is implemented as a simple stateless service that may be run on any server. In the course of a single transaction, a user may therefore interact with instances of the same service that are running on several different servers.
2. Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request). This allows the application to carry on doing useful work while it is waiting for the interaction to finish.
3. Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
4. Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.

20.7 Key Points

- The benefits of distributed systems are that they can be scaled to cope with increasing demand, can continue to provide user services (even if some parts of the system fail), and they enable resources to be shared.
- Issues to be considered in the design of distributed systems include transparency, openness, scalability, security, quality of service, and failure management.
- Client–server systems are distributed systems in which the system is structured into layers, with the presentation layer implemented on a client computer.
- Servers provide data management, application, and database services.
- Client–server systems may have several tiers, with different layers of the system distributed to different computers.
- Architectural patterns for distributed systems include master-slave architectures, two-tier and multitier client–server architectures, distributed component architectures, and peer-to-peer architectures.
- Distributed component systems require middleware to handle component communications and to allow components to be added to and removed from the system.

- Peer-to-peer architectures are decentralized architectures in which there are no distinguished clients and servers. Computations can be distributed over many systems in different organizations.
- Software as a service is a way of deploying applications as thin client–server systems, where the client is a web browser.

20.8 Bibliography

1. **‘Software Engineering’**, Ian Somerville, Pearson Education. Ninth Edition.
2. **‘Software Engineering’**, Roger Pressman, Tata-Mcgraw Hill, Seventh Edition.
3. Bernstein, P. A. (1996). **‘Middleware: A Model for Distributed System Services’**. Comm. ACM, 39 (2), 86–97.
4. Coulouris, G., Dollimore, J. and Kindberg, T. (2005). **‘Distributed Systems: Concepts and Design’**, 4th edition. Harlow, UK.: Addison-Wesley.
5. Holdener, A. T. (2008). **‘Ajax: The Definitive Guide. Sebastopol’**, Calif.: O’Reilly and Associates.
6. McDougall, P. (2000). **‘The Power of Peer-To-Peer’**. Information Week (August 28th, 2000).
7. Neuman, B. C. (1994). **‘Scale in Distributed Systems’**. In Readings in Distributed Computing Systems.
8. Casavant, T. and Singal, M. (ed.). Los Alamitos, Calif.: IEEE Computer Society Press.
9. Oram, A. (2001). **‘Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology’**.
10. Orfali, R. and Harkey, D. (1998). **‘Client/server Programming with Java and CORBA’**. New York: John Wiley & Sons.
11. Orfali, R., Harkey, D. and Edwards, J. (1997). **‘Instant CORBA’**. Chichester, UK: John Wiley & Sons.
12. Pope, A. (1997). **‘The CORBA Reference Guide: Understanding the Common Request Broker Architecture’**. Boston: Addison-Wesley.
13. Tanenbaum, A. S. and Van Steen, M. (2007). **‘Distributed Systems: Principles and Paradigms’**, 2nd edition. Upper Saddle River, NJ: Prentice Hall.

20.8 Exercises

- 20.1 What do you understand by ‘scalability ’? Discuss the differences between ‘scaling up’ and ‘scaling out’ and explain when these different approaches to scalability may be used.
- 20.2 Explain why distributed software systems are more complex than centralized software systems, where all of the system functionality is implemented on a single computer.
- 20.3 Using an example of a remote procedure call, explain how middleware coordinates the interaction of computers in a distributed system.
- 20.4 What is the fundamental difference between a fat-client and a thin-client approach to client–server systems architectures?
- 20.5 You have been asked to design a secure system that requires strong authentication and authorization. The system must be designed so that communications between parts of the system cannot be intercepted and read by an attacker. Suggest the most appropriate client–server architecture for this system and, giving reasons for your answer, propose how functionality should be distributed between the client and the server systems.
- 20.6 Your customer wants to develop a system for stock information where dealers can access information about companies and evaluate various investment scenarios using a simulation system. Each dealer uses this simulation in a different way, according to his or her experience and the type of stocks in question. Suggest client–server architecture for this system that shows where functionality is located. Justify the client–server system model that you have chosen.
- 20.7 Using a distributed component approach, propose architecture for a national theater booking system. Users can check seat availability and book seats at a group of theaters. The system should support ticket returns so that people may return their tickets for last-minute resale to other customers.
- 20.8 Give two advantages and two disadvantages of decentralized and semicentralized peer-to-peer architectures.
- 20.9 Explain why deploying software as a service can reduce the IT support costs for a company. What additional costs might arise if this deployment model is used?
- 20.10 Your company wishes to move from using desktop applications to accessing the same functionality remotely as services. Identify three risks that might arise and suggest how these risks may be reduced.
