DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS
2019 SCHEME

CST 309 MANAGEMENT OF SOFTWARE SYSTEMS

VISION OF THE INSTITUTION

To mold true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged

ABOUT DEPARTMENT

Established in: 2002

Course offered: B.Tech in Computer Science and Engineering

M.Tech in Computer Science and Engineering

M.Tech in Cyber Security

Approved by AICTE New Delhi and Accredited by NAAC

Affiliated to the University of Dr. A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.
5.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.
PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.
PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.
PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamwork and leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance

Optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

C305.1	Demonstrate Traditional and Agile Software Development approaches.	K3
C305.2	Prepare Software Requirement Specification and Software Design for a given problem.	K3
C305.3	Justify the significance of design patterns and licensing terms in software development, prepare testing, maintenance and DevOps strategies for a project	K3
C305.4	Demonstrate use of software project management concepts while planning, estimation, scheduling, tracking and change management of a project, with a traditional/agile framework.	K3
C305.5	Utilize SQA practices, Process Improvement techniques and Technology advancements in cloud based software models and containers & microservices.	K3

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES & PSO

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C305.1	3	3	3	3		3						3
C305.2	3	3	3	3		3				3	3	3
C305.3	3	3	3	3				3		3	3	3
C305.4	3	3	3	3		3			3	3	3	3
C305.5	3	3	3	3		3						3

CO PSO Mapping

CO'S	PSO1	PSO2	PSO3
C305.1	✓	✓	
C305.2	✓	✓	
C305.3	✓	✓	
C305.4	✓	✓	
C305.5	✓	✓	

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

CST 309	MANAGEMENT OF SOFTWARE SYSTEMS	Category	L	T	P	Credit	Year of Introduction
		PCC	3	0	0	3	2019

Syllabus

Module 1 : Introduction to Software Engineering (7 hours)

Introduction to Software Engineering - Professional software development, Software engineering ethics. Software process models - The waterfall model, Incremental development. Process activities - Software specification, Software design and implementation, Software validation, Software evolution. Coping with change - Prototyping, Incremental delivery, Boehm's Spiral Model. Agile software development - Agile methods, agile manifesto - values and principles. Agile development techniques, Agile Project Management. Case studies : An insulin pump control system. Mentcare - a patient information system for mental health care.

Module 2 : Requirement Analysis and Design (8 hours)

Functional and non-functional requirements, Requirements engineering processes. Requirements elicitation, Requirements validation, Requirements change, Traceability Matrix. Developing use cases, Software Requirements Specification Template, Personas, Scenarios, User stories, Feature identification. Design concepts - Design within the context of software engineering, Design Process, Design concepts, Design Model. Architectural Design - Software Architecture, Architectural Styles, Architectural considerations, Architectural Design Component level design - What is a component?, Designing Class-Based Components, Conducting Component level design, Component level design for web-apps. Template of a Design Document as per "IEEE Std 1016-2009 IEEE Standard for Information Technology Systems Design Software Design Descriptions". Case study: The Ariane 5 launcher failure.

Module 3 : Implementation and Testing (9 hours)

Object-oriented design using the UML, Design patterns, Implementation issues, Open-source development - Open-source licensing - GPL, LGPL, BSD. Review Techniques - Cost impact of Software Defects, Code review and statistical analysis. Informal Review, Formal Technical Reviews, Post-mortem evaluations. Software testing strategies - Unit Testing, Integration Testing, Validation testing, System testing, Debugging, White box testing, Path testing, Control Structure testing, Black box testing, Testing Documentation and Help facilities. Test automation, Test-driven development, Security testing. Overview of DevOps and Code Management - Code management, DevOps automation, Continuous Integration, Delivery, and Deployment (CI/CD/CD). Software Evolution - Evolution processes, Software maintenance.

Module 4 : Software Project Management (6 hours)

Software Project Management - Risk management, Managing people, Teamwork. Project Planning, Software pricing, Plan-driven development, Project scheduling, Agile planning. Estimation techniques, COCOMO cost modeling. Configuration management, Version management, System building, Change management, Release management, Agile software management - SCRUM framework. Kanban methodology and lean approaches.

Module 5 : Software Quality, Process Improvement and Technology trends (6 hours)

Software Quality, Software Quality Dilemma, Achieving Software Quality Elements of Software Quality Assurance, SQA Tasks , Software measurement and metrics. Software Process Improvement(SPI), SPI Process CMMI process improvement framework, ISO 9001:2000 for Software. Cloud-based Software - Virtualisation and containers, Everything as a service(IaaS, PaaS), Software as a service. Microservices Architecture - Microservices, Microservices architecture, Microservice deployment.

Text Books

1. Book 1 - Ian Sommerville, Software Engineering, Pearson Education, Tenth edition, 2015.
2. Book 2 - Roger S. Pressman, Software Engineering : A practitioner’s approach, McGraw Hill publication, Eighth edition, 2014
3. Book 3 - Ian Sommerville, Engineering Software Products: An Introduction to Modern Software Engineering, Pearson Education, First Edition, 2020.

References

1. IEEE Std 830-1998 - IEEE Recommended Practice for Software Requirements SpeciÞcations
2. IEEE Std 1016-2009 IEEE Standard for Information Technology—Systems Design—Software Design DescriptionsDavid J. Anderson, Kanban, Blue Hole Press 2010
3. David J. Anderson, Agile Management for Software Engineering, Pearson, 2003
4. Walker Royce, Software Project Management : A unified framework, Pearson Education, 1998
5. Steve. Denning, The age of agile, how smart companies are transforming the way work gets done. New York, Amacom, 2018.
6. Satya Nadella, Hit Refresh: The Quest to Rediscover Microsoft's Soul and Imagine a Better Future for Everyone, Harper Business, 2017
7. Henrico Dolfing, Project Failure Case Studies: Lessons learned from other people's mistakes, Kindle edition
8. Mary Poppendieck, Implementing Lean Software Development: From Concept to Cash, Addison-Wesley Signature Series, 2006
9. StarUML documentation - https://docs.staruml.io/
10. OpenProject documentation - https://docs.openproject.org/
11. BugZilla documentation - https://www.bugzilla.org/docs/
12. GitHub documentation - https://guides.github.com/
13. Jira documentation - https://www.atlassian.com/software/jira

Model Question Paper

QP CODE:

Reg No:_____

Name :_____ **PAGES : 3**

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

FIFTH SEMESTER B.TECH DEGREE EXAMINATION, MONTH & YEAR

Course Code: CST 309

Course Name: Management of Software Systems

Duration: 3 Hrs **Max. Marks :100**

PART A

Answer all Questions. Each question carries 3 marks

1. Why professional software that is developed for a customer is not simply the programs that have been developed and delivered.
2. Incremental software development could be very effectively used for customers who do not have a clear idea about the systems needed for their operations. Justify.
3. Identify any four types of requirements that may be defined for a software system
4. Describe software architecture
5. Differentiate between GPL and LGPL?
6. Compare white box testing and black box testing.
7. Specify the importance of risk management in software project management?
8. Describe COCOMO cost estimation model.
9. Discuss the software quality dilemma
10. List the levels of the CMMI model? **(10x3=30)**

Part B

(Answer any one question from each module. Each question carries 14 Marks)

11. (a) Compare waterfall model and spiral model **(8)**

(b) Explain Agile ceremonies and Agile manifesto **(6)**

12. (a) Illustrate software process activities with an example. **(8)**

(b) Explain Agile Development techniques and Agile Project Management **(6)**

13. (a) What are functional and nonfunctional requirements? Imagine that you are developing a library management software for your college, list eight functional requirements and four nonfunctional requirements. **(10)**

(b) List the components of a software requirement specification? **(4)**

OR

14. (a) Explain Personas, Scenarios, User stories and Feature identification? **(8)**

(b) Compare Software Architecture design and Component level design **(6)**

15. (a) Explain software testing strategies. **(8)**

(b) Describe the formal and informal review techniques. **(6)**

OR

16. (a) Explain Continuous Integration, Delivery, and Deployment CI/CD/CD) **(8)**

(b) Explain test driven development **(6)**

17. (a) What is a critical path and demonstrate its significance in a project schedule with the help of a sample project schedule. **(8)**

(b) Explain plan driven development and project scheduling. **(6)**

OR

18. (a) Explain elements of Software Quality Assurance and SQA Tasks. **(6)**

(b) What is algorithmic cost modeling? What problems does it suffer from when **(8)**

compared with other approaches to cost estimation?

19. (a) Explain elements of Software Quality Assurance and SQA Tasks. **(8)**

(b) Illustrate SPI process with an example. **(6)**

OR

20. (a) Compare CMMI and ISO 9001:2000. **(8)**

(b) How can Software projects benefit from Container deployment and Micro service deployment? **(6)**

Teaching Plan

No	Contents	No of Lecture Hrs
Module 1 : Introduction to Software Engineering (7 hours)		
1.1	Introduction to Software Engineering.[Book 1, Chapter 1]	1 hour
1.2	Software process models [Book 1 - Chapter 2]	1 hour
1.3	Process activities [Book 1 - Chapter 2]	1 hour
1.4	Coping with change [Book 1 - Chapter 2, Book 2 - Chapter 4]	1 hour
1.5	Case studies : An insulin pump control system. Mentcare - a patient information system for mental health care. [Book 1 - Chapter 1]	1 hour
1.6	Agile software development [Book 1 - Chapter 3]	1 hour
1.7	Agile development techniques, Agile Project Management.[Book 1 - Chapter 3]	1 hour
Module 2 : Requirement Analysis and Design (8 hours)		
2.1	Functional and non-functional requirements, Requirements engineering processes [Book 1 - Chapter 4]	1 hour
2.2	Requirements elicitation, Requirements validation, Requirements change, Traceability Matrix [Book 1 - Chapter 4]	1 hour
2.3	Developing use cases, Software Requirements Specification Template [Book 2 - Chapter 8]	1 hour

2.4	Personas, Scenarios, User stories, Feature identification [Book 3 - Chapter 3]	1 hour
2.5	Design concepts [Book 2 - Chapter 12]	1 hour
2.6	Architectural Design [Book 2 - Chapter 13]	1 hour
2.7	Component level design [Book 2 - Chapter 14]	1 hour
2.8	Design Document Template. Case study: The Ariane 5 launcher failure. [Ref - 2, Book 2 - Chapter 16]	1 hour
Module 3 : Implementation and Testing (9 hours)		
3.1	Object-oriented design using the UML, Design patterns [Book 1 - Chapter 7]	1 hour
3.2	Implementation issues, Open-source development - Open-source licensing - GPL, LGPL, BSD [Book 1 - Chapter 7]	1 hour
3.3	Review Techniques - Cost impact of Software Defects, Code review and statistical analysis. [Book 2 - Chapter 20]	1 hour
34	Informal Review, Formal Technical Reviews, Post-mortem evaluations. [Book 2 - Chapter 20]	1 hour
3.5	Software testing strategies - Unit Testing, Integration Testing, Validation testing, System testing and Debugging (basic concepts only). [Book 2 - Chapter 22]	1 hour
3.6	White box testing, Path testing, Control Structure testing, Black box testing. Test documentation [Book 2 - Chapter 23]	1 hour
3.7	Test automation, Test-driven development, Security testing. [Book 3 - Chapter 9]	1 hour
3.8	DevOps and Code Management - Code management, DevOps automation, CI/CD/CD. [Book 3 - Chapter 10]	1 hour
3.9	Software Evolution - Evolution processes, Software maintenance. [Book 1 - Chapter 9]	1 hour
Module 4 : Software Project Management (6 hours)		
4.1	Software Project Management - Risk management, Managing people, Teamwork [Book 1 - Chapter 22]	1 hour
4.2	Project Planning - Software pricing, Plan-driven development, Project scheduling, Agile planning [Book 1 - Chapter 23]	1 hour
4.3	Estimation techniques [Book 1 - Chapter 23]	1 hour
4.4	Configuration management [Book 1 - Chapter 25]	1 hour

4.5	Agile software management - SCRUM framework [Book 2 - Chapter 5]	1 hour
4.6	Kanban methodology and lean approaches.[Ref 9 - Chapter 2]	1 hour
Module 5 : Software Quality, Process Improvement and Technology trends (6 hours)		
5.1	Software Quality, Software Quality Dilemma, Achieving Software Quality. [Book 2 - Chapter 19]	1 hour
5.2	Elements of Software Quality Assurance, SQA Tasks , Software measurement and metrics. [Book 3 - Chapter 21]	1 hour
5.3	Software Process Improvement (SPI), SPI Process [Book 2 - Chapter 37]	1 hour
5.4	CMMI process improvement framework, ISO 9001:2000 for Software. [Book 2 - Chapter 37]	1 hour
5.5	Cloud-based Software - Virtualisation and containers, IaaS, PaaS, SaaS.[Book 3 - Chapter 5]	1 hour
5.6	Microservices Architecture - Microservices, Microservices architecture, Microservice deployment [Book 3 - Chapter 6]	1 hour

QUESTION PAPER

SL. .NO.	QUESTIONS	KL/CO
	MODULE 1	
1	Write the advantages of an incremental development model over a waterfallmodel?	K3/CO1
2	Illustrate how the process differs in agile software development and traditionalsoftware development with a socially relevant case study.	K3/CO1
	MODULE 2	
3	How to prepare a software requirement specification?	K2/CO2
4	Differentiate between Architectural design and Component level design.	K4/CO2
5	How does agile approaches help software developers to capture and define the userrequirements effectively?	K2/CO2
6	Write the relevance of the SRS specification in software development?	K3/CO2
7	Prepare a use case diagram for a library management system.	K6/CO2
	MODULE 3	
8	Differentiate between the different types of software testing strategies.	K4/CO3
9	Justify the need for DevOps practices?	K5/CO3
10	How do design patterns help software architects communicate the design of a complexsystem effectively?	K2/CO3
11	Write the proactive approaches one can take to optimise efforts in the testing phase?	K3/CO3
	MODULE 4	
12	Illustrate the activities involved in software project management for a socially relevantproblem?	K3/CO4
13	How do SCRUM, Kanban and Lean methodologies help software project management?	K2/CO4
14	Is rolling level planning in software project management beneficial? Justify your answer.	K5/CO4
15	How would you assess the risks in your software development project? Explain how you can manage identified risks?	K2/CO4

MODULE 5		
16	Justify the importance of Software Process improvement?	K5/CO5
17	Explain the benefits of cloud based software development, containers and microservices.	K2/CO5
18	Give the role of retrospectives in improving the software development process.	K2/CO5
19	Illustrate the use of project history data as a prediction tool to plan future socially relevant projects.	K2/CO5

MODULE NOTES

MODULE 1 NOTES

MODULE 1 : Introduction to Software Engineering (7 hours)

- **Introduction to Software Engineering - Professional software development, Software engineering ethics**
- **Software process models - The waterfall model, Incremental development. Process activities - Software specification, Software design and implementation, Software validation, Software evolution. Coping with change - Prototyping, Incremental delivery, Boehm's Spiral Model.**
- **Agile software development - Agile methods, agile manifesto - values and principles. Agile development techniques, Agile Project Management.**
- **Case studies: An insulin pump control system. Mentcare - a patient information system for mental health care.**

1.1 Professional software development

- Software is not just a program themselves but also all associated documentation and configuration data.

Frequently asked questions about software engineering

Question	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.
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 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 key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
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 system. 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.

Software Products

- Generic products
  - Stand-alone systems that are marketed and sold to any customer who wishes to buy them.
  - Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
  - Organization that develops the software controls the software specification.

Customized products(bespoke)

- Software that is commissioned by a specific customer to meet their own needs.
- Examples – embedded control systems, air traffic control software, traffic monitoring systems.
- Specification is developed and controlled by the organization ie buying the software.

Essential Attributes of Good Software

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 utilisation, 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.

1.1.1 Software Engineering

- Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- Engineering discipline
    - Using appropriate theories and methods to solve problems within the organizational and financial constraints.
- All aspects of software production
    - Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

Software Process Activities

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

General issues that affect most Software

- Heterogeneity
    - Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.
- 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 to rapidly develop new software.
- Security and trust

- As software is intertwined with all aspects of our lives, it is essential that we can trust that software.

1.1.2 *Software Engineering Diversity*

- There are many different types of software system and there is no universal set of software techniques that is applicable to all of these.
- The software engineering methods and tools used depend on the type of application being developed, the requirements of the customer and the background of the development team.

Application Types

- Stand-alone applications
  - These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.
- Interactive transaction-based applications
  - Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.
- Embedded control systems
  - These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.
- Batch processing systems
  - These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.
- Entertainment systems
  - These are systems that are primarily for personal use and which are intended to entertain the user.
- Systems for modeling and simulation
  - These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting

objects.

- Data collection systems
    - These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.
- Systems of systems
    - These are systems that are composed of a number of other software systems.
    - software that has already been developed rather than write new software.

1.1.3 *Software Engineering and the Web*

- The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.
- Web services allow application functionality to be accessed over the web.
- Cloud computing is an approach to the provision of computer services where applications run remotely on the 'cloud'.
    - Users do not buy software buy pay according to use.

Web software Engineering

- Software reuse is the dominant approach for constructing web-based systems.
    - When building these systems, you think about how you can assemble them from pre-existing software components and systems.
- Web-based systems should be developed and delivered incrementally.
    - It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.
- User interfaces are constrained by the capabilities of web browsers.
    - Technologies such as AJAX allow rich interfaces to be created within a web browser but are still difficult to use. Web forms with local scripting are more commonly used.

Web based Software Engineering

- Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.
- The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software in the same way that they apply to other types of software system.

1.2 *Software Engineering Ethics*

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

Issues of Professional Responsibility

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

ACM/IEEE Code of Ethics

- The professional societies in the US have cooperated to produce a code of ethical practice.
- Members of these organisations sign up to the code of practice when they join.
- The Code contains eight Principles related to the behavior of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

ACM/IEEE Code of Ethics

- Software Engineering Code of Ethics and Professional Practice

- ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices
- PREAMBLE
- The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.
- Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Ethical principles

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

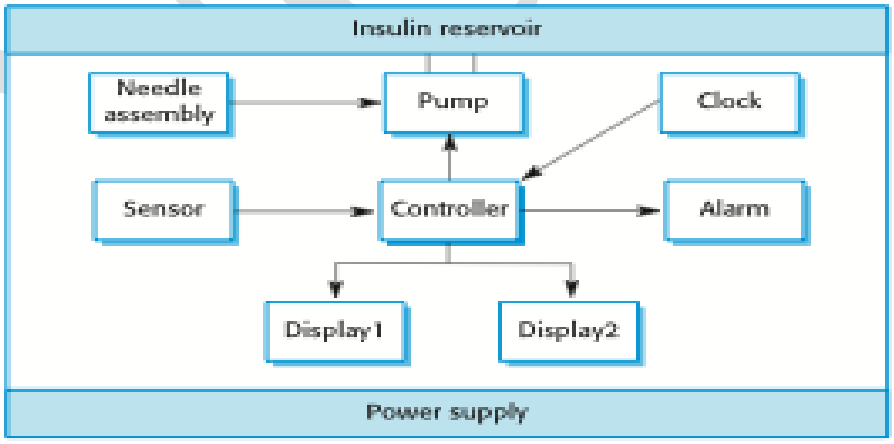
1.3 Case Studies

- A personal insulin pump
    - An embedded system in an insulin pump used by diabetics to maintain blood glucose control.
- A mental health case patient management system
    - A system used to maintain records of people receiving care for mental health problems.
- A wilderness weather station
    - A data collection system that collects data about weather conditions in remote areas.

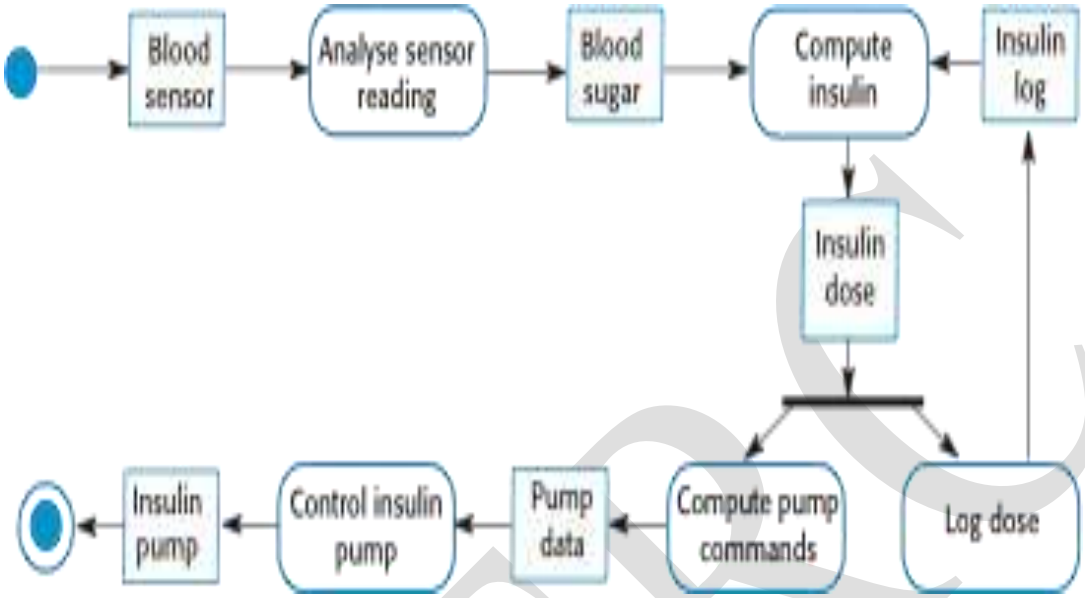
1.3.1 *Insulin Pump Control System*

- Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
- Calculation based on the rate of change of blood sugar levels.
- Sends signals to a micro-pump to deliver the correct dose of insulin.
- Safety-critical system as low blood sugars can lead to brain malfunctioning, coma and death; high-blood sugar levels have long-term consequences such as eye and kidney damage.

Insulin Pump Hardware Architecture



Activity model of the insulin pump



Essential High-Level Requirements

- The system shall be available to deliver insulin when required.
- The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.
- The system must therefore be designed and implemented to ensure that the system always meets these requirements.

1.3.2 *A Patient Information System for Mental Health Care*

- A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.

- To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centres.

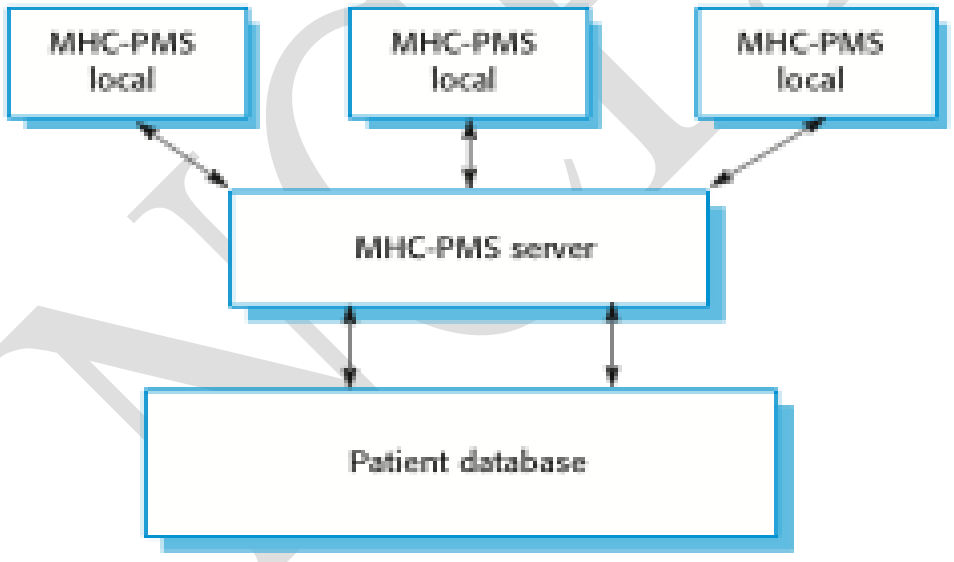
MHC-PMS

- The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics.
- It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected.

MHC-PMS goals

- To generate management information that allows health service managers to assess performance against local and government targets.
- To provide medical staff with timely information to support the treatment of patients.

The organization of the MHC-PMS



MHC-PMS Key Features

- Individual care management
  - Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors can quickly learn about the key problems and treatments that have been prescribed.

- Patient monitoring
    - The system monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.
- Administrative reporting
    - The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

MHC-PMS concerns

- Privacy
    - It is essential that patient information is confidential and is never disclosed to anyone apart from authorised medical staff and the patient themselves.
- Safety
    - Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
    - The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe the correct medication to patients.

1.3.3 *Wilderness Weather Station*

- The government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.
- Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
    - The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes

parameter readings periodically and manages the data collected from the instruments.

The Weather Station's Environment

Weather information system

- The weather station system

  This is responsible for collecting weather data, carrying out some initial data processing and transmitting it to the data management system.

- The data management and archiving system

  This system collects the data from all of the wilderness weather stations, carries out data processing and analysis and archives the data.

- The station maintenance system

  This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems.

Additional software functionality

- Monitor the instruments, power and communication hardware and report faults to the management system.
- Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
- Support dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

The Software Process

- A structured set of activities required to develop a software system.
- Many different software processes but all involve:
    - Specification – defining what the system should do;
    - Design and implementation – defining the organization of the system and implementing the system;
    - Validation – checking that it does what the customer wants;
    - Evolution – changing the system in response to changing customer needs.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Software Process Descriptions

- When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- Process descriptions may also include:
    - **Products**, which are the outcomes of a process activity;
    - **Roles,** which reflect the responsibilities of the people involved in the process;
    - **Pre- and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced.

Plan-Driven and Agile Processes

- Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- In practice, most practical processes include elements of both plan-driven and agile approaches.
- There are no right or wrong software processes.

Software Process Models

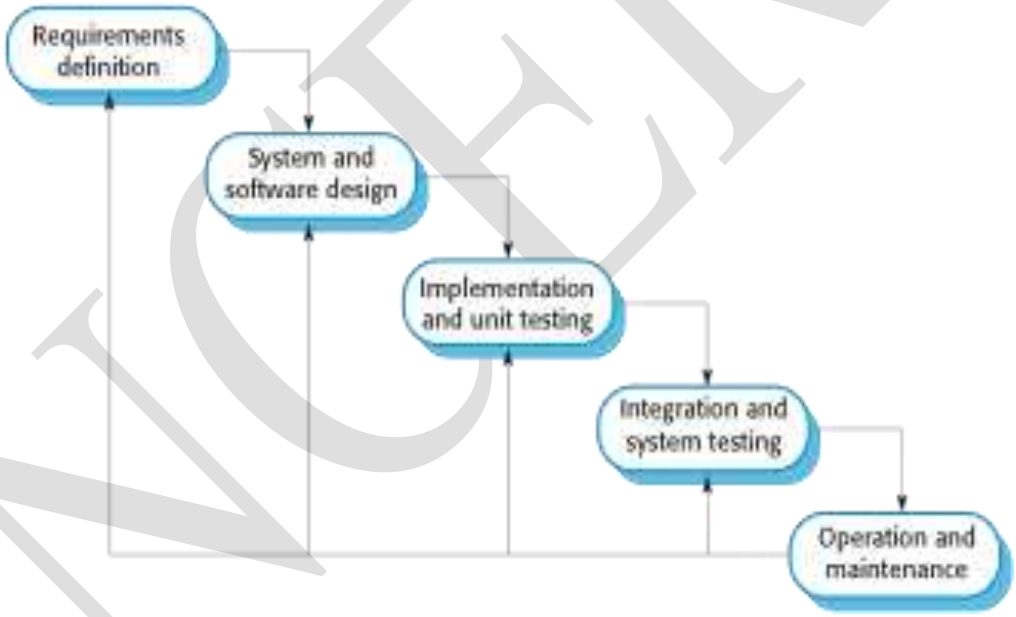
- The waterfall model

- Plan-driven model. Separate and distinct phases of specification and development.

- ***Incremental development***
  - Specification, development and validation are interleaved. May be plan-driven or agile.
- ***Reuse-oriented software engineering(Operation and maintenance)***
  - The system is assembled from existing components. May be plan-driven or agile.
- In practice, most large systems are developed using a process that incorporates elements from all of these models.

The Waterfall Model

Waterfall model phases



There are separate identified phases in the waterfall model:

- **Requirements analysis and definition**: The system services, constraints and goals are established by consultation with system users.
- **System and software design**: The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and

describing the fundamental software system abstractions and their relationships.

- **Implementation and unit testing**: During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its multiplication.
- **Integration and system testing**: The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system isdelivered to the customer.
- **Operation and maintenance**: longest phase, the system is installed and put into the practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improves the implementation of system units and enhancing the system's services as new requirements are discovered.

The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

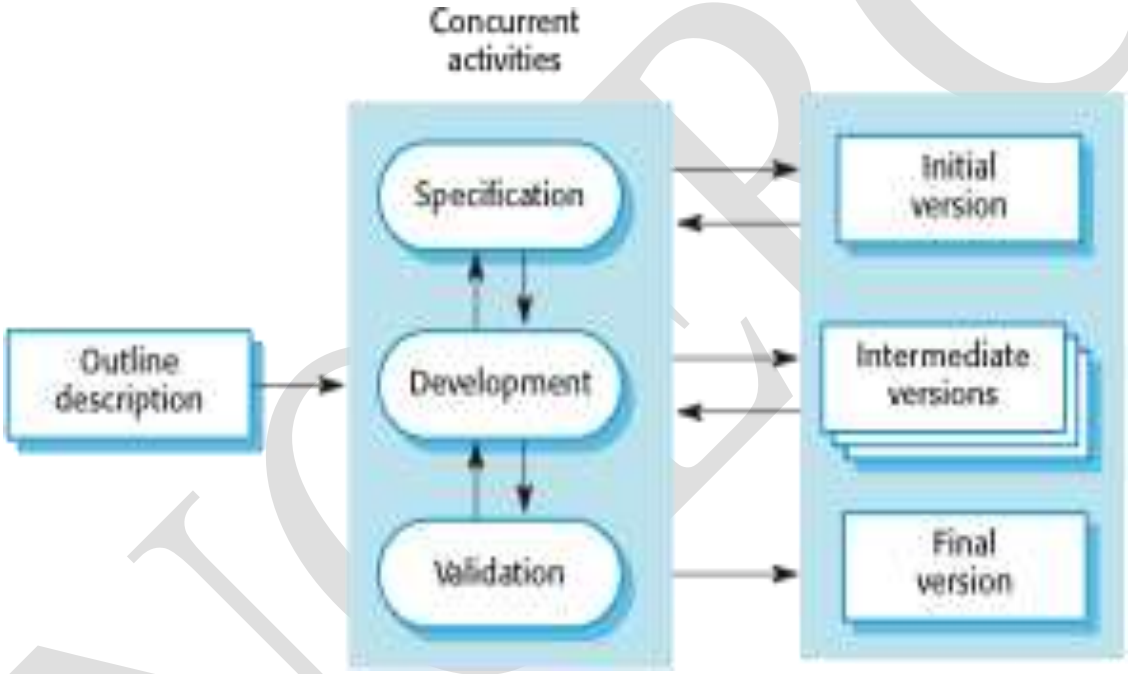
Waterfall Model Problems

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
    - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
    - Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
    - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

Incremental Development

Incremental Development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed. Specification, development and validation activities are interleaved rather than separate, with rapid feedback across activities.

Each increment of the system incorporates some functionality that is needed by the customer. This means that the customer can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed, and new functionality defined for later increments.



Incremental Development Benefits

- The cost of accommodating changing customer requirements is reduced.
    - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- It is easier to get customer feedback on the development work that has been done.
    - Customers can comment on demonstrations of the software and see how much has been implemented.
- More rapid delivery and deployment of useful software to the customer is possible.

- Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental Development Problems

- The process is not visible.
    - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added.
    - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

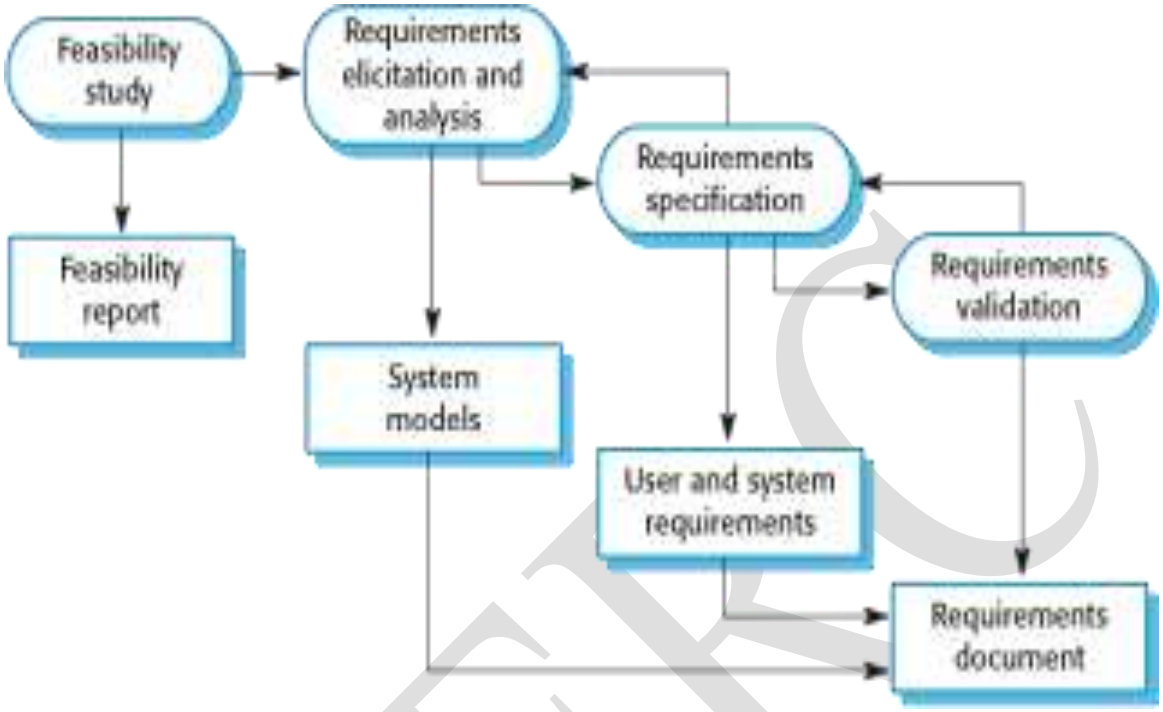
Process Activities

- Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- The four basic process activities of Specification, Development, Validation and Evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are inter-leaved.

1. Software Specification

- The process of establishing and defining what services are required from the system and identifying the constraints on the system's operation and development.
- Is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in system design and implementation.
- RE process aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements.

Requirements are presented at two levels: End users and customers need a high level statement of the requirements; system developers need a more detailed system specification



Requirements engineering process

- Feasibility study
  - Is it technically and financially feasible to build the system?
  - Developed within the existing budgetary constraints.(cost effective)
- Requirements elicitation and analysis
  - What do the system stakeholders require or expect from the system?
  - Observations from existing systems, discussions with potential users, task analysis.
  - This may involve the development of one or more models and prototypes
- Requirements specification
  - Is the activity of translating the information gathered during the analysis activity into a document.

Two types of requirements

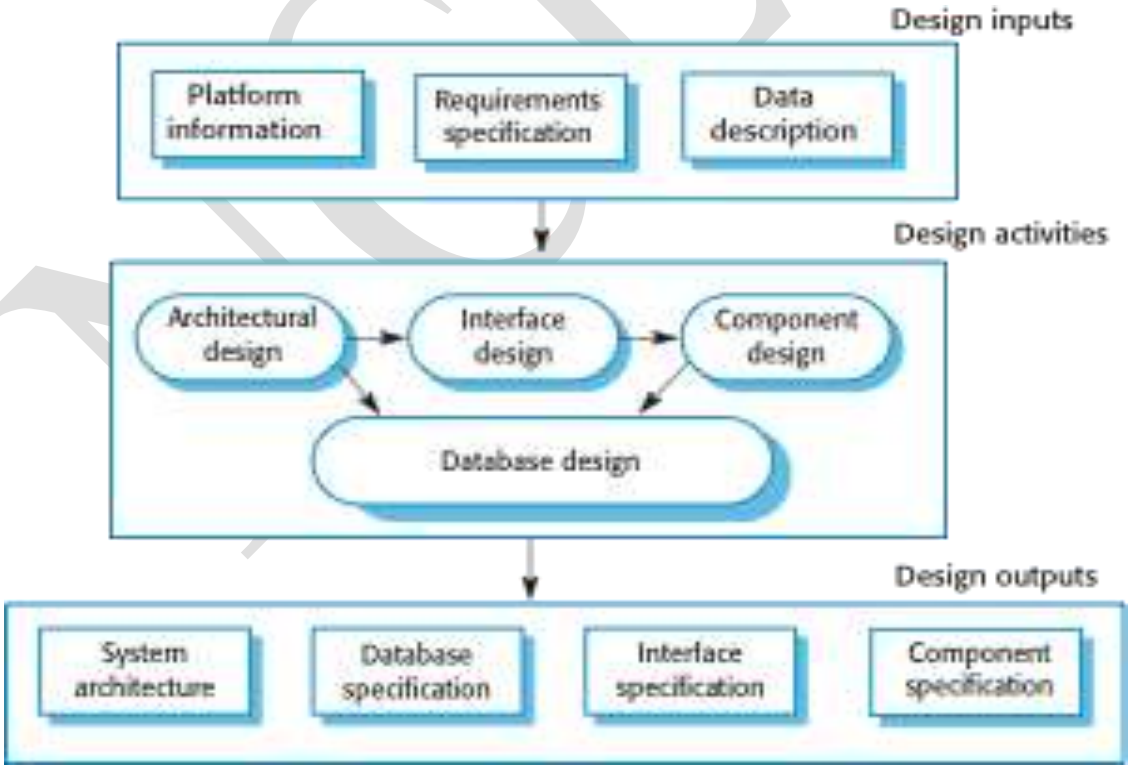
- User requirements: are abstract statements of the system requirements for the customer and end user of the system.
- System requirements are a more detailed description of the functionality to be provided.

- Requirements validation
  - Checking the validity of the requirements(consistent/complete)

2. *Software Design and Implementation*

- The process of converting the system specification into an executable system.
- Software design
  - Design a software structure that realises the specification;
- Implementation
  - Translate this structure into an executable program;
- The activities of design and implementation are closely related and may be inter-leaved.

A General Model of the Design Process



- **Software platform**-the environment in which software will execute.
- Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with the software's environment.
- The **requirement specification** is the description of the functionality the software must provide and its performance and dependability requirements.
- If the system is to process existing data, then the **description of that data** may be included in the platform specification.
- Otherwise, the data description must be an input to the design process so that the system data organization to be defined.

Design Activities

- **Architectural design**, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.
- **Interface design**, where you define the interfaces between system components. This interface specification must be unambiguous
- **Component design,** where you take each system component and design how it will operate.
- **Database design**, where you design the system data structures and how these are to be represented in a database. The work depends on whether an existing database is to be reused or a new database is to be created.

3. Software Validation

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking processes such as inspections and reviews.
- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- Testing is the most commonly used V & V activity.

Stages of Testing

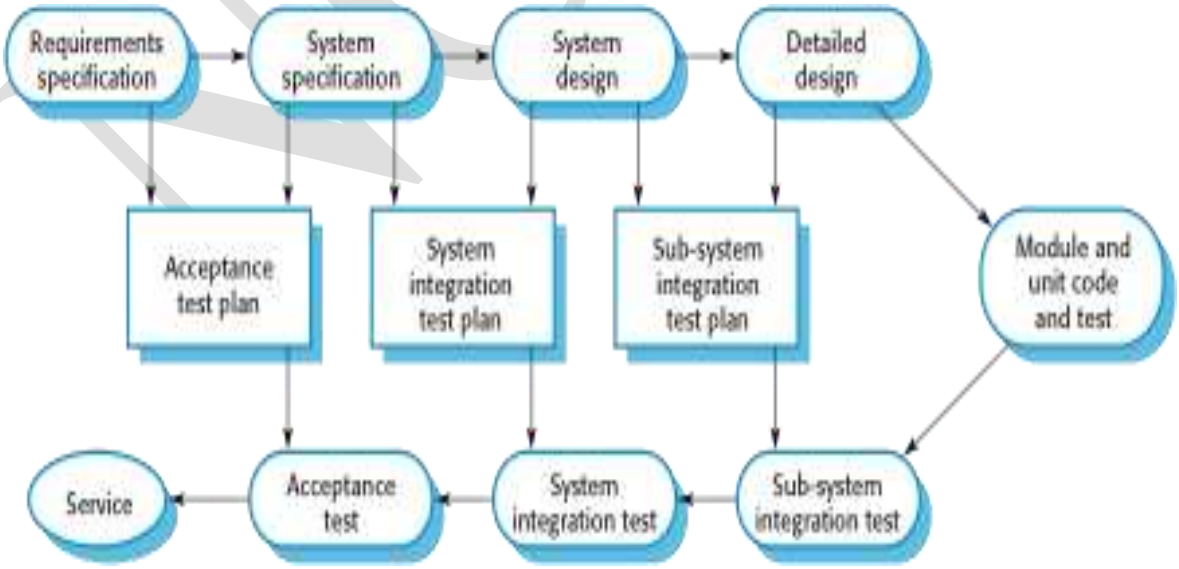
3 stage process

System components are tested (component defects are discovered early in the process) then the integrated system is tested, (interface problems are found when the system is integrated), finally the system is tested with the customer's data.

Testing Stages

- Development or component testing
  - Individual components are tested independently;
  - Components may be functions or objects or coherent groupings of these entities.
  - Test automation tools such as JUnit that can rerun component tests when new versions of the components are created, are commonly used.
- System testing
  - Testing of the system as a whole.
  - Concerned with showing the system meets its functional and non-functional requirements, Testing of emergent properties is particularly important.
- Acceptance testing(alpha testing)
  - This is the final stage in the testing process before the system is accepted for operational use.
  - The system is tested with data supplied by the system customer rather than with simulated test data .Testing with customer data to check that the system meets the customer's needs.

Testing Phases in a Plan-Driven Software Process



- Acceptance testing(alpha testing)
  - Alpha Testing is a type of software testing performed to identify bugs before releasing the product to real users or to the public. Alpha Testing is one of the user acceptance testing.
  - Custom systems are developed for a single client
  - This alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of requirements.

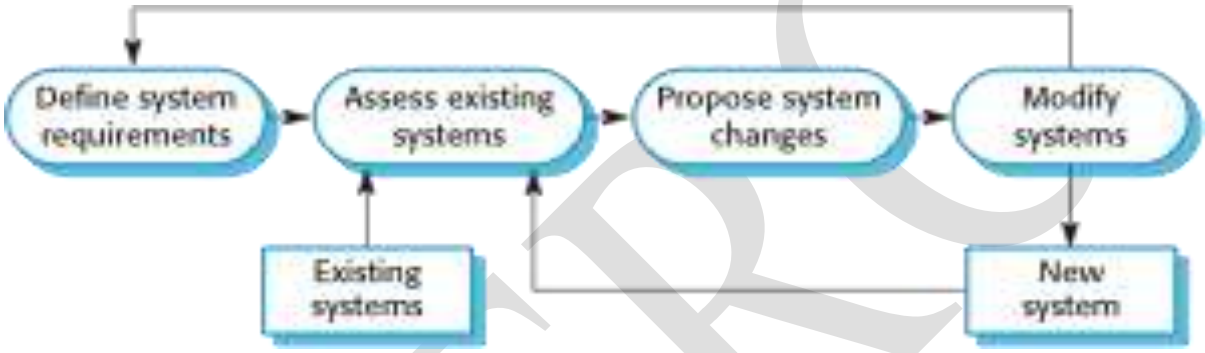
Beta testing

- When software is to be marketed as a software product, beta testing is used.
- Beta Testing is performed by real users of the software application in a real environment.
- This involves delivering a system to a number of potential users who agree to use that system.
- They report problem to system developers.
- This exposes the product to real use and detects errors that may not have been anticipated by the system builders.
- After this feedback, the system is modified and released either for further beta testing or general sale.

Alpha Testing	Beta Testing
Alpha testing involves both the white box and black box testing.	Beta testing commonly uses black box testing.
Alpha testing is performed by testers who are usually internal employees of the organization.	Beta testing is performed by clients who are not part of the organization.
Alpha testing is performed at developer's site.	Beta testing is performed at end-user of the product.
Reliability and security testing are not checked in alpha testing.	Reliability, security and robustness are checked during beta testing.
Alpha testing ensures the quality of the product before forwarding to beta testing.	Beta testing also concentrates on the quality of the product but collects users input on the product and ensures that the product is ready for real time users.
Alpha testing requires a testing environment or a lab.	Beta testing doesn't require a testing environment or lab.
Alpha testing may require long execution cycle.	Beta testing requires only a few weeks of execution.
Developers can immediately address the critical issues or fixes in alpha testing.	Most of the issues or feedback collected from beta testing will be implemented in future versions of the product.

4. Software Evolution

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.



Coping with change

- Change is inevitable in all large software projects.
  - Business changes lead to new and changed system requirements
  - New technologies open up new possibilities for improving implementations
  - Changing platforms require application changes
- Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality.

Reducing the Costs of Rework

- **Change avoidance**, where the software process includes activities that can anticipate possible changes before significant rework is required.
  - For example, a prototype system may be developed to show some key features of the system to customers.
- **Change tolerance**, where the process is designed so that changes can be accommodated at relatively low cost.
  - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed.

If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.

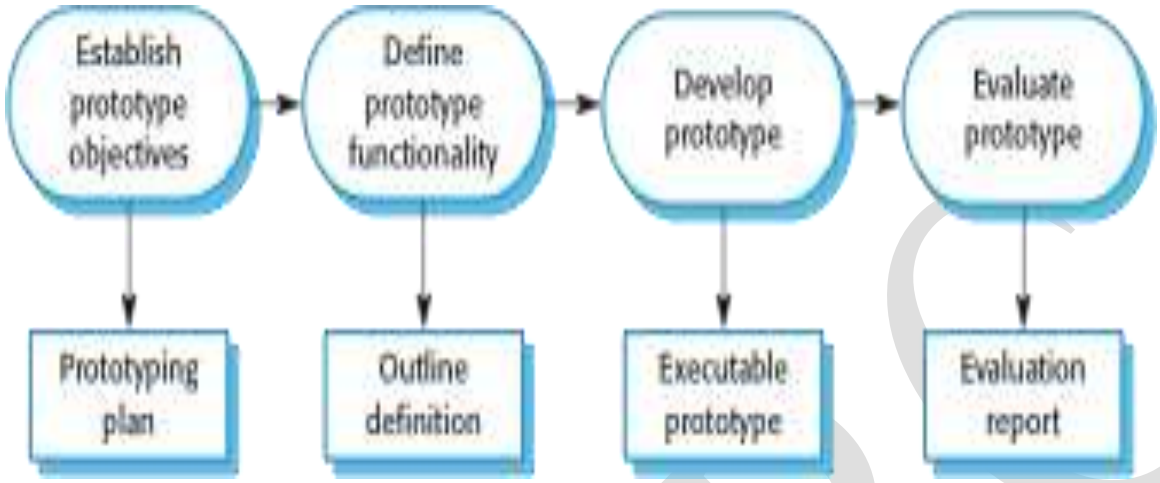
Software Prototyping

- A prototype is an initial version of a system used to demonstrate concepts and try out design options, and find out more about the problem and its possible solutions.
- Where a version of the system or part of the system is developed quickly to check the customer requirements.
- Rapid, iterative development of the prototype is essential, so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.
- A prototype can be used in:
    - The requirements engineering process can help with requirements elicitation and validation;
- In design processes to explore particular software solutions options and develop a UI design;

Benefits of Prototyping

- Improved system usability.
- A closer match to users' real needs.
- Improved design quality.
- Improved maintainability.
- Reduced development effort.

The Process of Prototype Development



- The objectives of prototyping should be made explicit from the start of the process. This may develop a system to prototype the user interface, or to validate he functional requirements, to demonstrate the feasibility of the application to managers. The same prototype cannot meet all objcetives.so they misunderstand the functionality of the prototype development. - May be based on rapid prototyping languages or tools.
- The second stage is to decide what to put into /leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, leave some functionality out of the prototype-May be some nonfunctional requirements. Focus on functional rather than non-functional requirements such as reliability and security
- Prototype should focus on areas of the product that are not well-understood;
- Error checking and recovery may not be included in the prototype;
- Final stage is evaluation.

Developers are pressured by managers to deliver Throw away prototypes, when there are delays in delivering the final version of the software.

Throw-Away Prototypes

- Prototypes should be discarded after development as they are not a good basis for a production system:
    - It may be impossible to tune the system to meet non-functional requirements; such as performance ,security ,robustness

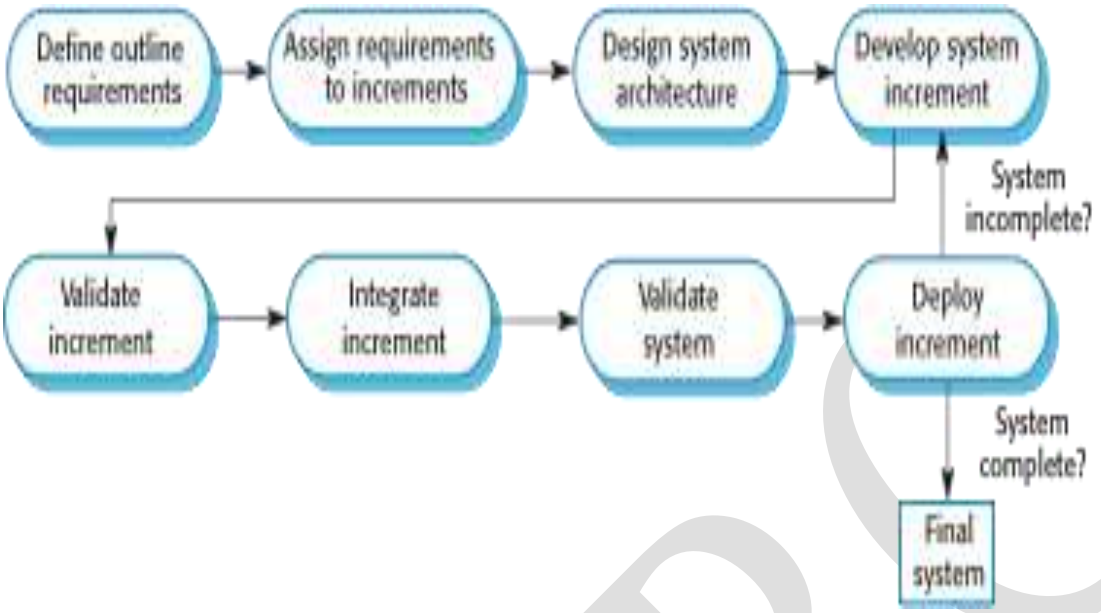
- Prototypes are normally undocumented; only design specification is prototype code. This is not good enough for long term maintenance.
- The prototype structure is usually degraded through rapid change;the system will be difficult and expensive to maintain,.
- The prototype probably will not meet normal organizational quality standards.

Incremental Delivery

- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- In an incremental delivery process, customers define **which of the services** are most important.
- User requirements are prioritised and the highest priority requirements are included in early increments.
- Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place, but requirements changes for the current increment are not accepted.
- Once an increment is completed and delivered, it is installed in the customer's normal working environment. They can experiment with the system, and this helps them clarify their requirements for later system increments. As new increments are completed, they are integrated with existing increments so that system functionality improves with each delivered increment.

Incremental Delivery Advantages

- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.



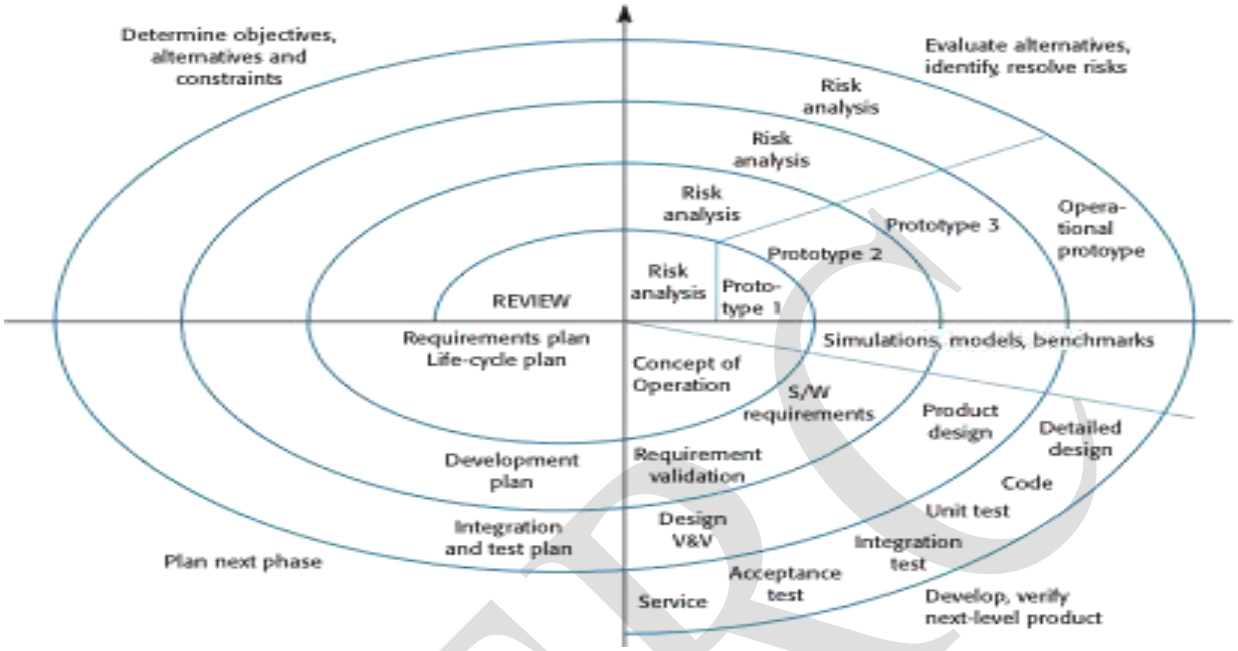
Incremental Delivery

Incremental Delivery Problems

- Most systems require a set of basic facilities that are used by different parts of the system.
    - As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- The essence of iterative processes is that the specification is developed in conjunction with the software.
    - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

Boehm's spiral model

- Process is represented as a spiral rather than as a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.



Each loop in the Spiral Model is split into 4 Sectors

- Objective setting
    - Specific objectives for the phase are identified. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies may planned.
- Risk assessment and reduction
    - Risks are assessed and activities put in place to reduce the key risks.
- Development and validation
    - A development model for the system is chosen which can be any of the generic models.
- Planning
    - The project is reviewed and the next phase of the spiral is planned.

Spiral Model Usage

- Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.
- In practice, however, the model is rarely used as published for practical software development.

Agile Software Development

Rapid software development became known as agile development or agile methods. Rapid development and delivery is now often the most important requirement for software systems

- Businesses operate in a fast changing requirement and it is practically impossible to produce a set of stable software requirements
- Software has to evolve quickly to reflect changing business needs.

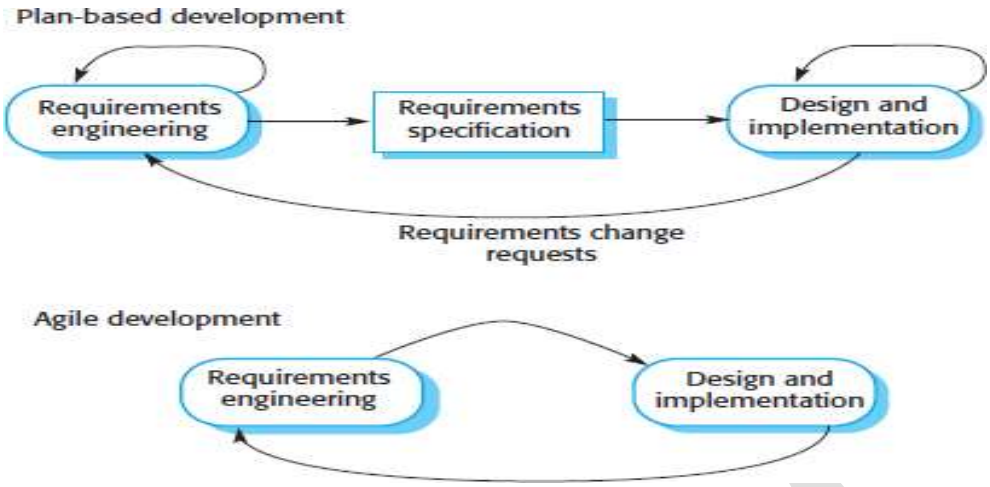
Agile development characteristics

- **Specification, design and implementation are inter-leaved**, there is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system.
- **System is developed as a series of versions with stakeholders involved in version evaluation.** They may propose changes to the software and new requirements that should be implemented in a later version of the system.
- **Extensive tool support is used to support the development process**. Tools that may be used include automated testing tools, tools to support configuration management, and system integration and tools to automate user interface production. User interfacesare often developed using an IDE and graphical toolset.

Agile methods are incremental development methods in which the increments are small, and, typically, new releases of the system are created (frequent release)and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

- In a plan-driven software development process, iteration occurs within activities, with formal documents used to communicate between stages of the process. For example, the requirements will evolve, and, ultimately, a requirements specification will be produced. This is then an input to the design and implementation process.

- In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together rather than separately



- ***Agile Methods***
  - Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
    - Focus on the code rather than the design
    - Are based on an iterative approach to software development
    - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.
  - The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

Principles of agile methods

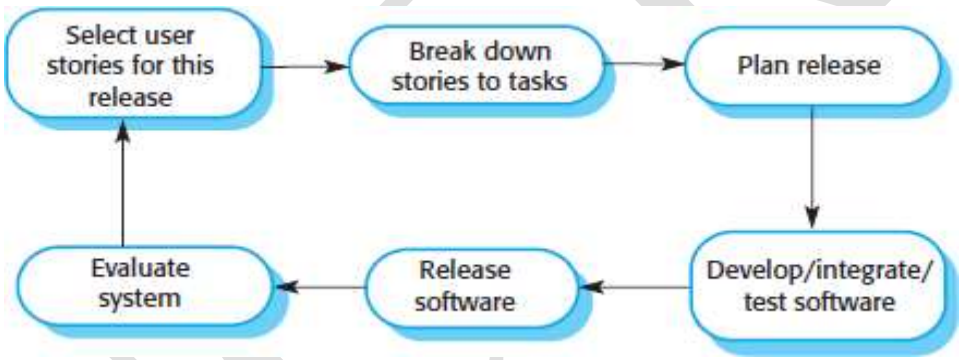
Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer 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.

Agile Manifesto

- 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.

Agile Development Techniques

XP: Figure illustrates the XP process an increment of the system that is being developed.



XP Release cycle

In XP, 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.
Extreme programming was an agile practices that were summarized and reflect the principles of the agile manifesto:

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.

XP programming practices

Some important practices used in the agile development (XP) are

User stories:

- Software requirements always change. In Agile methods , requirements elicitation is integrated with development by the idea of "**user stories**" where a user story is a scenario of use that might be experienced by a system user.
- After the discussion of development team with customer, they develop a "**story card**" that briefly describes a story that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software.
- User stories may be used in planning system iterations. Once the story cards have been developed, the development team breaks these down into tasks 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.

- 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.
- User stories can be helpful in getting users involved in suggesting requirements during an initial predevelopment requirements elicitation activity.

Cons:

- The principal problem with user stories is completeness. It is difficult to judge if enough user stories have been developed to cover all of the essential requirements of a system.

- It is also difficult to judge if a single story gives a true picture of an activity. Experienced users are often so familiar with their work that they leave things out when describing it.

Principle or practice	Description
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.
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." See Figures 3.5 and 3.6.
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.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
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.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
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:

- Changes will always have to be made to the code being developed. Refactoring means that the programming team look for possible improvements to the software and implements them immediately.
- Refactoring improves the software structure and readability and avoids the structural deterioration that naturally occurs when software is changed.

Test-first development:

Extreme Programming developed a new approach to program testing to address the difficulties of testing without a specification. Testing is automated and is central to the development process, and development cannot proceed until all tests have been successfully executed. The key features of testing in XP are:

1. test-first development:
   - Write test before write the code.
   - Writing tests implicitly defines both an interface and a specification ofbehaviour for the functionality being developed.
   - Problems of requirements and interface misunderstandings are reduced.
   - Test-first development requires there to be a clear relationship between system requirements and the code implementing the corresponding requirements.
   - In XP, this relationship is clear because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit ofimplementation.
   - In test-first development, the task implementers have to thoroughly understandthe 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. It also avoids the problem of "test- lag." This may happen when the developer of the system works at a faster pacethan the tester.
2. Incremental test development from scenarios,
   - Develop each tasks, so that the development schedule can be maintained.
3. User involvement in the test development and validation, and
   - The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
4. The use of automated testing frameworks.
   - 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 is a widely used example of an automated testing framework for Java programs.

Pair programming:

The programming pair sits at the same computer to develop the software. However, the same pair do not always program together. Rather, pairs are created dynamically so that allteam members work with each other during the development process.

Pair programming has a number of advantages.

1. **It supports the idea of collective ownership and responsibility for the system**. This reflects Weinberg's idea of egoless programming where the software is owned by the teamas 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 leasttwo people.

3. It encourages refactoring to improve the software structure.

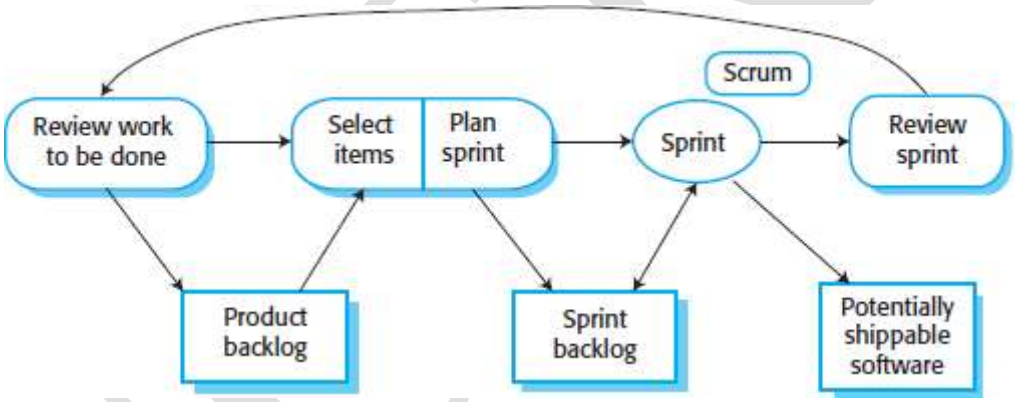
Agile Project Management

- The principal responsibility of software project managers is to manage the project sothat the software is delivered on time and within the planned budget for the project.

Scrum

- The Scrum approach is a general agile method and focus is on managing iterativedevelopment rather than specific agile practices.

The Scrum Process



The Sprint Cycle

- Each process iteration produces a product increment that could be delivered to customers.
- The starting point for planning is the **product backlog**, which is the list of work to be done on the project. —the list of items such as product features, requirements, user stories and engineering improvement that have to be worked on by the Scrum team.
- The product owner has a responsibility to ensure the level of specification is appropriatefor the work to be done.

- Each sprint cycle lasts a fixed length of time, which is usually between 2 and 4 weeks. At the beginning of each cycle, the Product Owner prioritizes the items on the product backlog to define which are the most important items to be developed in that cycle.
- Sprints are never extended to take account of unfinished work. Items are returned to the product backlog if these cannot be completed within the allocated time for the sprint.
- The whole team is then involved in selecting which of the highest priority items they believe can be completed. They then estimate the time required to complete these items. To make these estimates, they use the velocity attained in previous sprints, that is, howmuch of the backlog could be covered in a single sprint. This leads to the creation of a**sprint backlog**—the work to be done during that sprint.
- The team self-organizes to decide who will work on what, and the sprint begins.

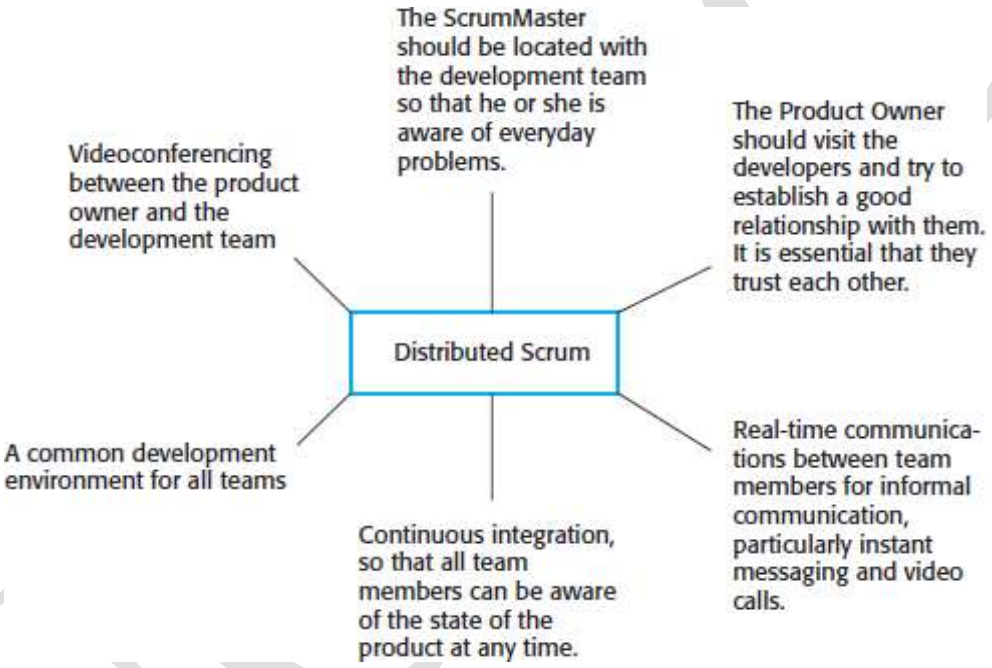
Teamwork in Scrum

- The 'Scrum master' is a facilitator who arranges daily meetings, tracks the backlog of work to be done, records decisions, measures progress against the backlog and communicates with customers and management outside of the team.
- The whole team attends short daily meetings (scrum)where all team members share information, describe their progress since the last meeting, problems that have arisen and what is planned for the following day.
- This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them, there is no top-down direction from theScrum Master.
- Everyone participates in this short-term planning; the daily interactions among Scrum teams may be coordinated using a Scrum board. This is an office whiteboard that includes information and post-it notes about the Sprint backlog, work done, unavailability of staff, and so on. This is a shared resource for the whole team, and anyone can change or move items on the board. It means that any team member can, at a glance, see what others are doing and what work remains to be done.
- At the end of each sprint, there is a review meeting, which involves the whole team. This meeting has two purposes. First, it is a means of process improvement. The teamreviews the way they have worked and reflects on how things could have been done better. Second, it provides input on the product and the product state for the product backlog review that precedes the next sprint.

Scrum Benefits

- The product is broken down into a set of manageable and understandable chunks.
- Unstable requirements do not hold up progress.
- The whole team have visibility of everything and consequently team communication isimproved.
- Customers see on-time delivery of increments and gain feedback on how the product works.
- Trust between customers and developers is established and a positive culture is createdin which everyone expects the project to succeed.

For offshore development, the product owner is in a different country from the development team, which may also be distributed. Figure shows the requirements for Distributed Scrum



Key Points

- 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.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team and the culture of the company developing the system.

- The Scrum method is an agile method that provides a project management framework.It is centred round a set of sprints, which are fixed time periods when a system increment is developed.
- Scaling agile methods for large systems is difficult. Large systems need up-front designand some documentation.

MODULE 2 NOTES

MODULE 2

Requirement Analysis and Design (8 hours)

Functional and non-functional requirements, Requirements engineering processes, Requirements elicitation, Requirements validation, Requirements change, Traceability matrix, Developing use cases, Software Requirements Specification Template, Personas, Scenarios, User stories, Feature identification.
Design concepts - Design within the context of software engineering, Design Process, Design concepts, Design Model.
Architectural Design - Software Architecture, Architectural Styles, Architectural considerations, Architectural Design.
Component level design - What is a component?, Designing Class-Based Components, Conducting Component level design, Component level design for web-apps.
Template of a Design Document as per "IEEE Std 1016-2009 IEEE Standard for Information Technology Systems Design Software Design Descriptions". Case study: The Arianne 5 launcher failure.

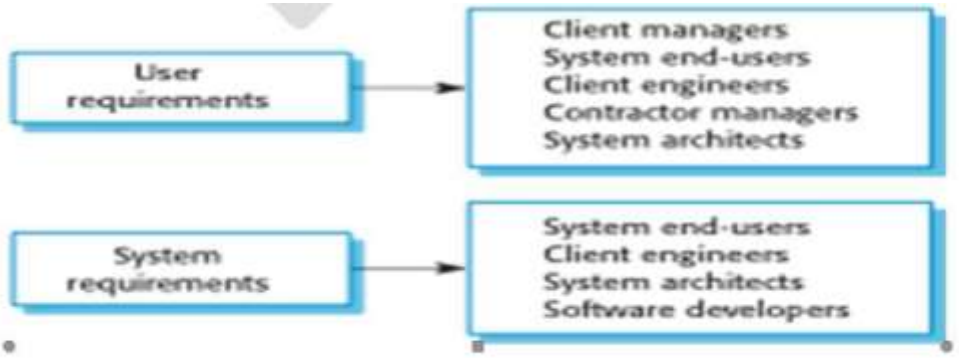
Requirements Engineering

The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

Types of requirement

- **User requirements-**
  Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- **System requirements-**
  A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor**.**

Readers of different types of requirements specification



Functional and non-functional requirements

- **Functional requirements-**
  Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  May state what the system should not do.
- **Non-functional requirements-**
  Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  Often apply to the system as a whole rather than individual features or services.
- **Domain requirements**
  Constraints on the system from the domain of operation

`

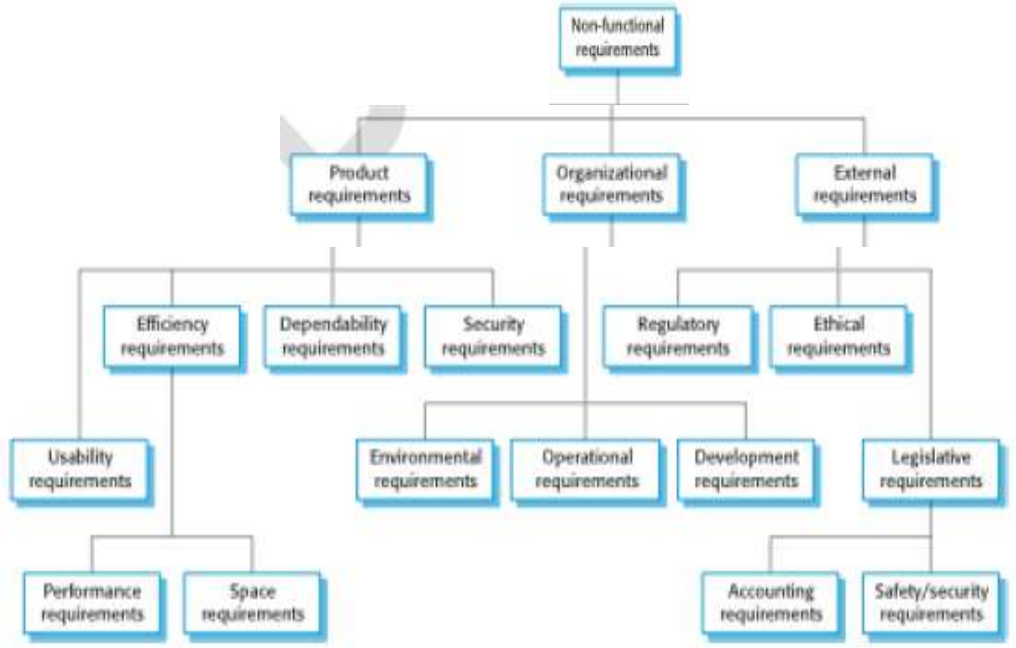
Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail**.**

Non-functional requirements

These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc. Process requirements may also be specified mandating a particular IDE, programming language or development method. Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

Types of nonfunctional requirement



Requirements which arise from factors which are external to the system development process e.g. interoperability requirements, legislative requirements etc.

- **Usability requirements**

  The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal). Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement).

Metrics for specifying nonfunctional requirements
Non-functional requirements may affect the overall architecture of a system rather than the individual components**.**

- **Non-functional classifications**
  - **Product requirements**
    Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
  - **Organizational requirements**
    Requirements which are a consequence of organizational policies and procedures e.g. process standards used, implementation requirements, etc.

- **External requirements**
  Requirementswhicharisefromfactorswhichareexternaltothesystemdevelopmentprocesse.g. interoperability requirements, legislative requirements etc.

- **Usability requirements**
  The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal). Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use. (Testable non-functional requirement).

Metricsforspecifyingnonfunctionalrequirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Requirements engineering processes
The processes used for Requirement Engineering vary widely depending on the application domain, the people involved and the organization developing the requirements. Requirement Engineering is an iterative activity in which these processes are interleaved.

A spiral view of the requirements engineering process

1. Requirements elicitation and analysis

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*

Problems of requirements analysis

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

Requirements elicitation and analysis

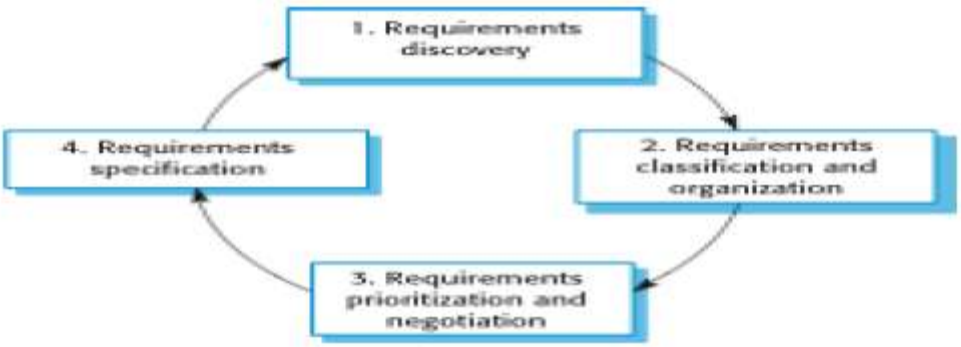
Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.

Stages include:

- **Requirements discovery and understanding:** process of interacting with stake holders to discover their requirements. Domain requirements from stakeholders and documentation are also discovered.

- **Requirements classification and organization:** this activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters**.**
- **Requirements prioritization and negotiation:** when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.
- **Requirements specification (documentation):** requirements are documented and input into the next round of spiral.

The requirements elicitation and analysis process



Requirements discovery (elicitation techniques)
The process of gathering information about the required and existing systems and distilling the user and system requirements from this information. Interaction is with system stakeholders from managers to external regulators. Systems normally have a range of stakeholders.

- **Interviewing**
  Formal or informal interviews with stakeholders are part of most RE processes. Formal or informal interviews with stakeholders are part of most RE processes. Types of interview
  - ✓ **Closed interviews :** stakeholders answers based on pre-determined list of questions
  - ✓ **Open interviews :** in which there is no predefined agenda, where various issues are explored with stakeholders

- **Effective interviewing**

  - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.

  - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

- **Scenarios**

Scenarios are real-life examples of how a system can be used. They should include

- A description of the starting situation;
- A description of the normal flow of events;
- A description of what can go wrong;
- Information about other concurrent activities;
- A description of the state when the scenario finishes.

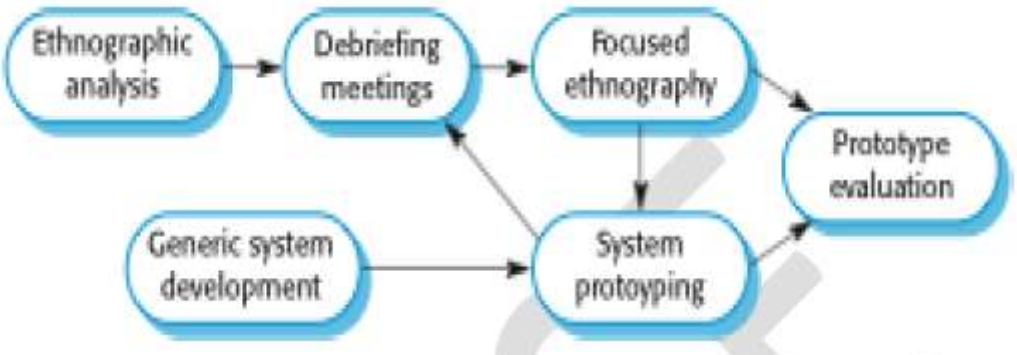
Ethnography

- A social scientist spends a considerable time observing and analyzing how people actually work. People do not have to explain or articulate their work.
- Social and organizational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.
- Requirements that are derived from cooperation and awareness of other people's activities.
- Awareness of what other people are doing leads to changes in the ways in which we do things.
- Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.

Focused ethnography

- Developed in a project studying the air traffic control process.
- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the ethnographic analysis.
- The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

Ethnography and prototyping for requirements analysis



2. Requirements specification

- The process of writing the user and system requirements in a requirements document.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
- **It is therefore important that these are as complete as possible. Ways of writing a system requirements specification**

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Requirements and design

In principle, requirements should state what the system should do and the design should describe how it does this.

In practice, requirements and design are inseparable

- A system architecture may be designed to structure the requirements;
- The system may inter-operate with other systems that generate design requirements;
- The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
- This may be the consequence of a regulatory requirement.

Natural language specification

- Requirements are written as natural language sentences supplemented by diagrams and tables.
- Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.
- Include an explanation (rationale) of why a requirement is necessary.

Problems with natural language

- Lack of clarity
- Precision is difficult without making the document difficult to read.
  Requirements confusion
- Functional and non-functional requirements tend to be mixed-up.
  Requirements amalgamation
- Several different requirements may be expressed together.

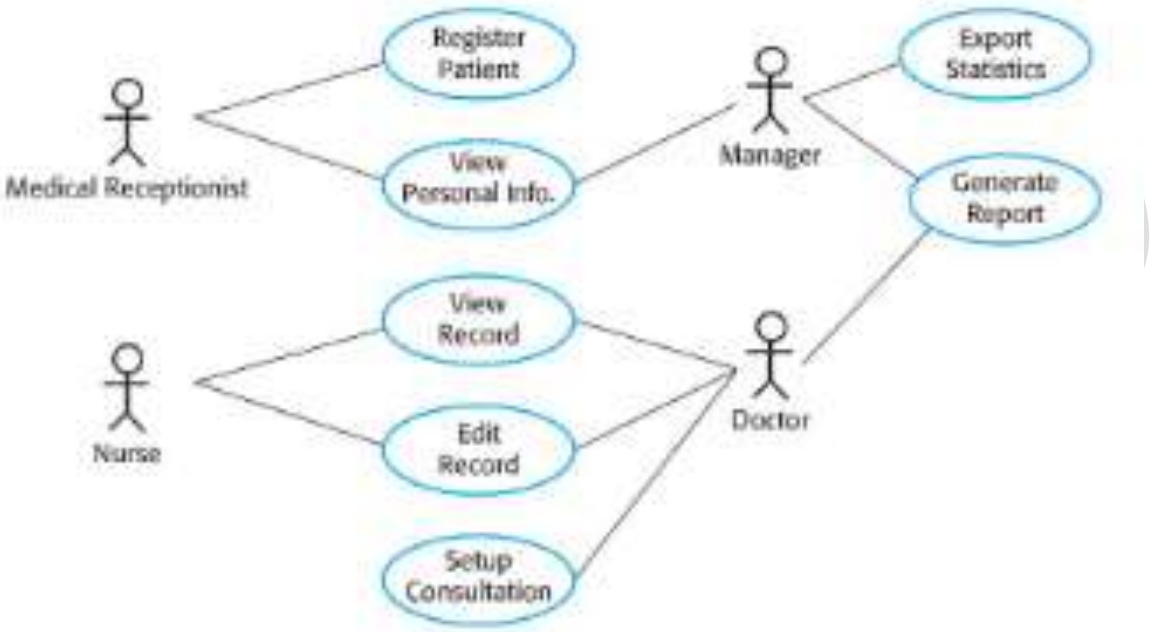
Structured specifications

- An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

Form-based specifications

- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Information about the information needed for the computation and other entities used.
- Description of the action to be taken.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

- **Tabular specification**
  - Used to supplement natural language.
  - Particularly useful when you have to define a number of possible alternative courses of action.
- **Use cases**
  - Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
  - A set of use cases should describe all possible interactions with the system.



- **Developing Use cases**
  - A use case tells a stylized story about how an end user (playing one of a number of possible roles) interacts with the system under a specific c set of circumstances.
  - The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation.
  - A use case depicts the software or system from the end user's point of view.
  - The first step in writing a use case is to define the set of "actors" that will be involved in the story.
  - Actors are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.

- Actors represent the roles that people (or devices) play as the system operates.
- An actor is anything that communicates with the system or product and that is external to the system itself.
- Primary actors → interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software.
- Secondary actors → support the system so that primary actors can do their work.
- Once actors have been identified, use cases can be developed

3. Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important.
- Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking

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

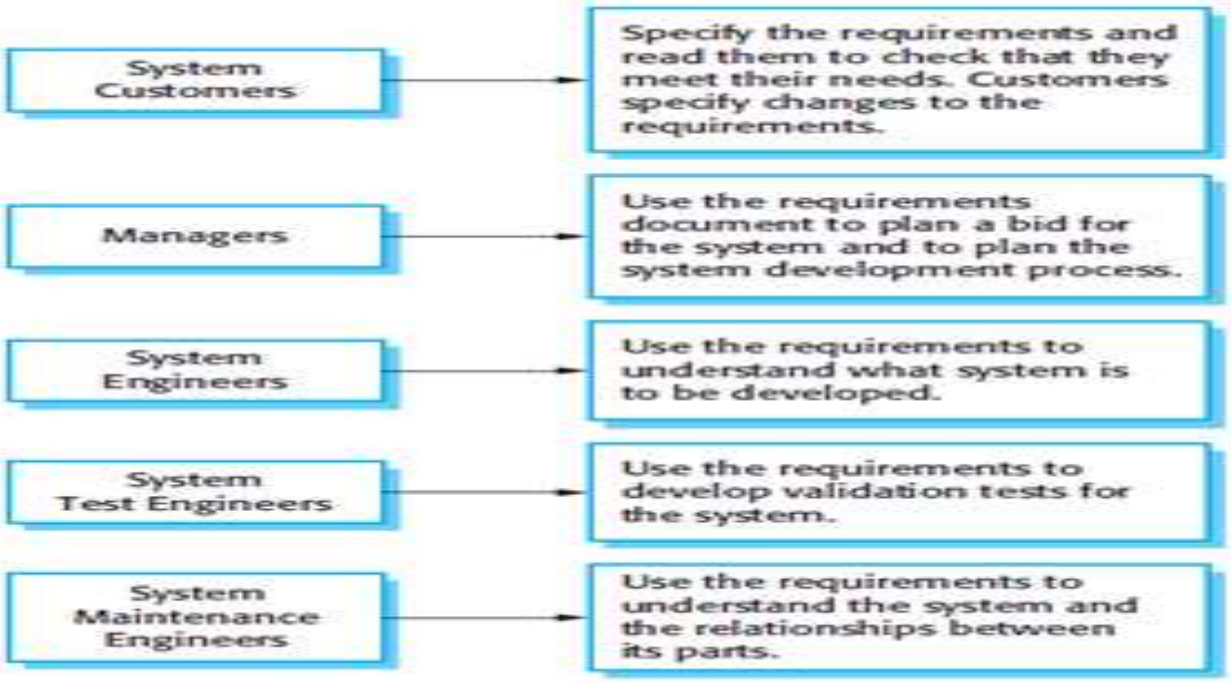
✓ **Requirements validation techniques**

- **Requirements reviews**
  Systematic manual analysis of the requirements: requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.

- **Prototyping**
  Using an executable model of the system to check requirements.
- **Test-case generation**
  Developing tests for requirements to check testability

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.



Users of a requirements document

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a work product that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents

Revision History

1. **Introduction**
   - 1.1 Purpose
   - 1.2 Document Conventions
   - 1.3 Intended Audience and Reading Suggestions
   - 1.4 Project Scope
   - 1.5 References
2. **Overall Description**
   - 2.1 Product Perspective
   - 2.2 Product Features
   - 2.3 User Classes and Characteristics
   - 2.4 Operating Environment
   - 2.5 Design and Implementation Constraints
   - 2.6 User Documentation
   - 2.7 Assumptions and Dependencies
3. **System Features**
   - 3.1 System Feature 1
   - 3.2 System Feature 2 (and so on)
4. **External Interface Requirements**
   - 4.1 User Interfaces
   - 4.2 Hardware Interfaces
   - 4.3 Software Interfaces
   - 4.4 Communications Interfaces
5. **Other Nonfunctional Requirements**
   - 5.1 Performance Requirements
   - 5.2 Safety Requirements
   - 5.3 Security Requirements
   - 5.4 Software Quality Attributes
6. **Other Requirements**

Appendix A: Glossary

Appendix B: Analysis Models

Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted in this sidebar.

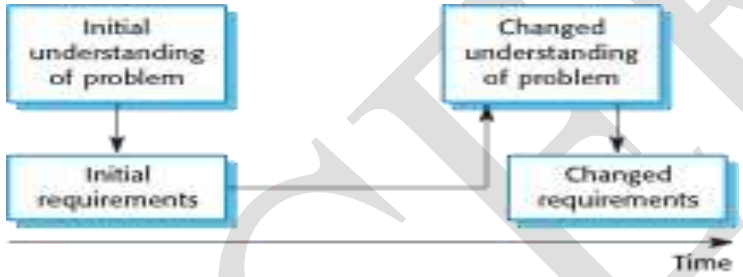
Requirements Management

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- New requirements emerge as a system is being developed and after it has gone into use.
- You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact **of** requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

✓ **Changing requirements**

- The business and technical environment of the system always changes after installation.
- The people who pay for a system and the users of that system are rarely the same people.
- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

✓ **Requirements evolution**



Requirements management planning

Establishes the level of requirements management detail that is required.Requirements management decisions:

- Requirements identification: Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
- A change management process: This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
- Traceability policies: These policies define the relationships between each requirement and between the requirements and the system design that should be recorded**.**
- Tool support: Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management

Deciding if a requirements change should be ccepted

Problem analysis and change specification

- During this stage, the problem or the change proposal is analyzed 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.

Change analysis and costing

- The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

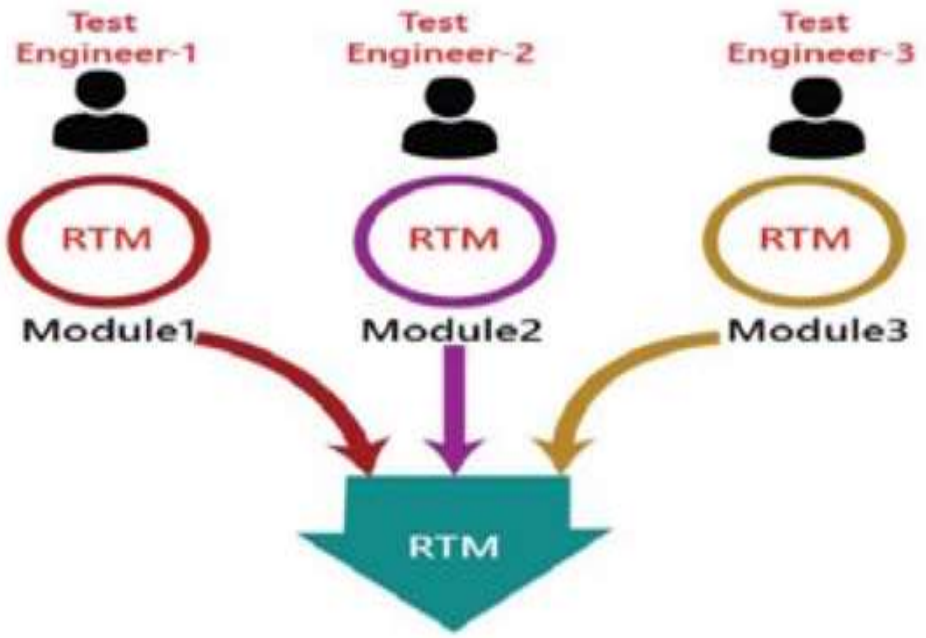
Change implementation

- The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented

Traceability Matrix

- Is an Engg team that refers to documented links between Software Engg work products (Eg Requirements and test cases)
- Traceability matrix allows a requirement engineer to represent the relationship between requirements and other work products.
- Rows of the matrix are labelled using requirement names and columns can be labelled with the name of Software Engg work product.
- A matrix cell is marked to indicate the presence of link between the two.
- A table type document that is used in the development of software application to trace requirements.
- It can be used for both forward (from Requirements to Design or Coding) and backward (from Coding to Requirements) tracing.
- It is also known as Requirement Traceability Matrix (RTM) or Cross Reference Matrix (CRM).
- It is prepared before the test execution process to ensure that every requirement is covered in the form of a Test case so that we don't miss out any testing.
- Map all the requirements and corresponding test cases to ensure that we have written all the test cases for each condition.

- This matrix can support a variety of Engg development activities.
- They can provide continuity for developers as a project moves from one project phase to another.
- It can be used to ensure the Engg work products have taken all requirements into account.
- As the no: of req and the number of work products grows.it become increasingly difficult to keep the traceability up to date.



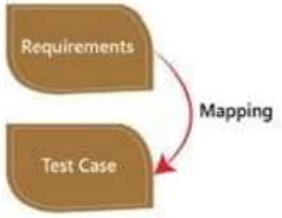
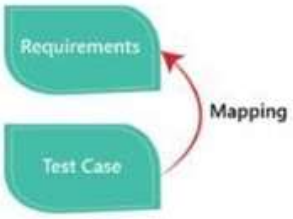
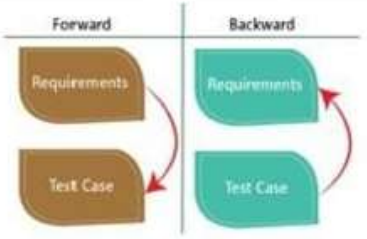
	A	B	C	D	E
1	RTM Template				
2	Requirement number	Module name	High level requirement	Low level requirement	Test case name
3	2	Loan	2.1 Personal loan	2.1.1--> personal loan for private employee	beta-2.0-personal loan
4				2.1.2--> personal loan for government employee	
5				2.1.3--> personal loan for jobless people	
6			2.2 Car loan	2.2.1--> car loan for private employee	
7				——	
8			2.3 Home loan	——	
9				——	
10				——	
11					

The traceability matrix can be classified into three different types which are as follows:

1. Forward traceability

2. Backward or reverse traceability

3. Bi-directional traceability

Forward Traceability	Backward Traceability	Bi-directional Traceability
• Used to ensure that every business's needs or requirements are executed correctly in the application and also tested rigorously. • Requirements are mapped into the forward direction to the test cases.	• Used to check that we are not increasing the space of the product by enhancing the design elements, code, test other things which are not mentioned in the business needs. • Requirements are mapped into the backward direction to the test cases.	• A combination of forwarding and backward traceability matrix. • Used to make sure that all the business needs are executed in the test cases. • Also evaluates the modification in the requirement which is occurring due to the bugs in the application.
		

Goals of Traceability Matrix:

• It helps in tracing the documents that are developed during various phases of SDLC.

• It ensures that the software completely meets the customer's requirements.

• It helps in detecting the root cause of any bug.

✓ Advantages of RTM:

• With the help of the RTM document, we can display the complete test execution and bugs status based on requirements.

• It is used to show the missing requirements or conflicts in documents.

• We can ensure the complete test coverage, which means all the modules are tested.

• It will also consider the efforts of the testing teamwork towards reworking or reconsidering on the test cases.

Personas, Scenarios and Stories ,Feature Identification

From personas to features

Fig: personas, scenarios, and user stories lead to features that might be implemented in a software product.

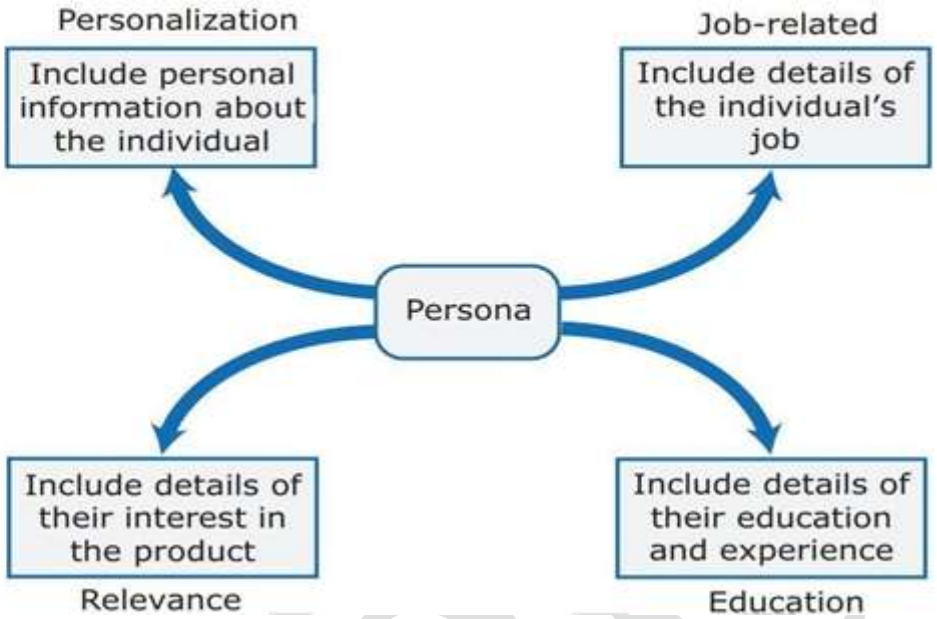
PERSONAS

- Personas are about "imagined users," character portraits of types of user that you think might adopt your product.
- Ex: if your product is aimed at managing appointments fordentists, you might create a dentist persona, a receptionist persona, and a patient persona.
- Personas of different types of users help to imagine what these users may want to do with your software and how they might use it.
- They also help you envisage difficulties that users might have in understanding and using product features.
- There is no standard way to represent personas

Persona should include the following:

- ✓ Description about the users' backgrounds
- ✓ Description about why the users might want to use your product
- ✓ Description about their education and technical skills.
- Personas should be relatively short and easy to read.
- Personas are a tool that allows team members to "step into the users' shoes." Instead of thinking about what they would do in a particular situation, they can imagine how a persona would behave and react.
- They can help you check your ideas to ensure that you are not including product features that aren't really needed.
- They help you to avoid making unwarranted assumptions, based on your own knowledge, and designing an overcomplicated or irrelevant product.
- Personas, scenarios and user stories lead to features that might be implemented in a software product.

Figure 3.4 Persona descriptions

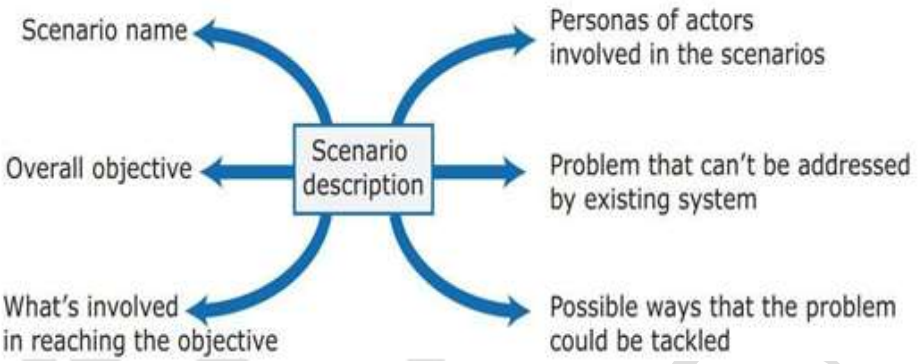


SCENARIO

Scenario is a narration that describes a situation in which a user is using your product's features to do something that they want to do.

- Scenarios are used in the design of requirements and system features, in system testing, and in user interface design.
- It should briefly explain the user's problem and present an imagined way that the problem might be solved.
- Scenarios are high level stories of system use.
- They should describe a sequence of interactions with the system but should not include details of these interactions.
- They, are the basics for both use cases, which are extensively used in object oriented methods, and user stories, which are used in agile methods.
  - Like personas, they help developers to gain a shared understanding of the system that they are creating.
  - Scenarios are not specifications. They lack detail, they may be incomplete, and they may not represent all types of user interactions.

Figure 3.5 Elements of a scenario description



Structured scenarios should include different fields such as:

- ✓ what the user sees at the beginning of a scenario,
- ✓ a description of the normal flow of events,
- ✓ a description of what might go wrong, and so on.

At the early stages of product design, the scenarios be narrative rather than structured.

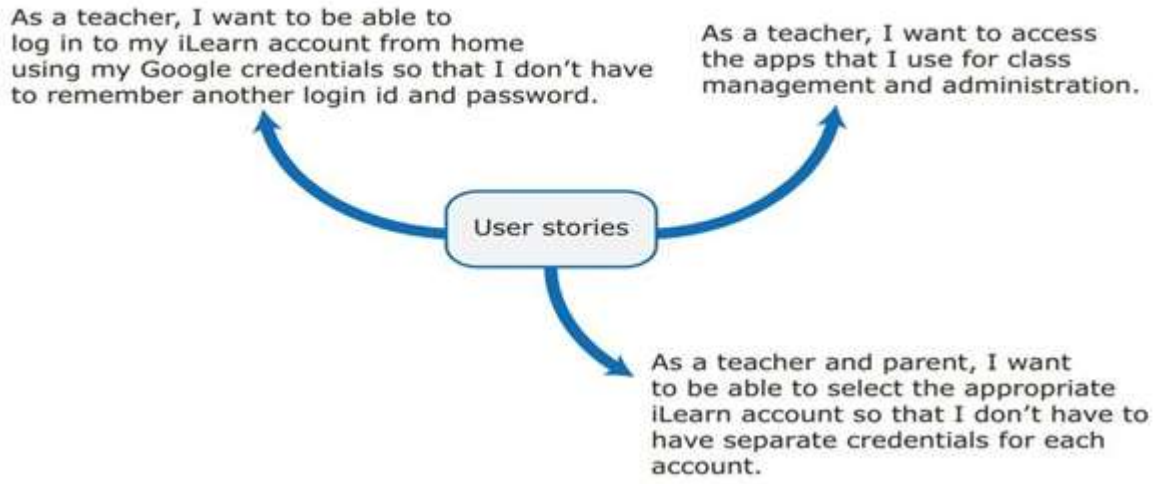
Writing scenarios

- ✓ Start with the personas that you have created.
- ✓ Tryto imagine several scenarios for each persona.
- ✓ Not necessary to include every details you think users might do with your product.
- ✓ Scenarios should always be written from the user's perspective and should be based on identified personas or real users.
- ✓ Scenario writing is not a systematic process and different teams approach it in different ways.
- ✓ Writing scenarios always gives you ideas for the features that you can include in the system.

User Stories

- ✓ These are finer-grain narratives that set out in a more detailed and structured way a single thing that a user wants from a software system.
- ✓ User stories are not intended for planning but for helping with feature identification.
- ✓ Aim to develop stories that are helpful in one of 2 ways:
- ✓ as a way of extending and adding detail to a scenario;
- ✓ as part of the description of the system feature that you have identified.

Figure 3.6 User stories from Emma's scenario



Feature Identification

- ✓ A feature is a way of allowing users to access and use your product's functionality so that the feature list defines the overall functionality ofthe system.
- ✓ Feature is a fragment of functionality that implements some user or system need. We can access features through user interface of a product.
- ✓ Feature is something that the user needs or wants.

Identify the product features that are independent, coherent and relevant:

- **Independence** ⟶ A feature should not depend on how other system features implemented and should not be affected by the order of activation of other features.
- **Coherence Features** ⟶ should be linked to a single item of functionality. They should not do more than one thing, and they should never have side effects.
- **Relevance System features** ⟶ should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required.

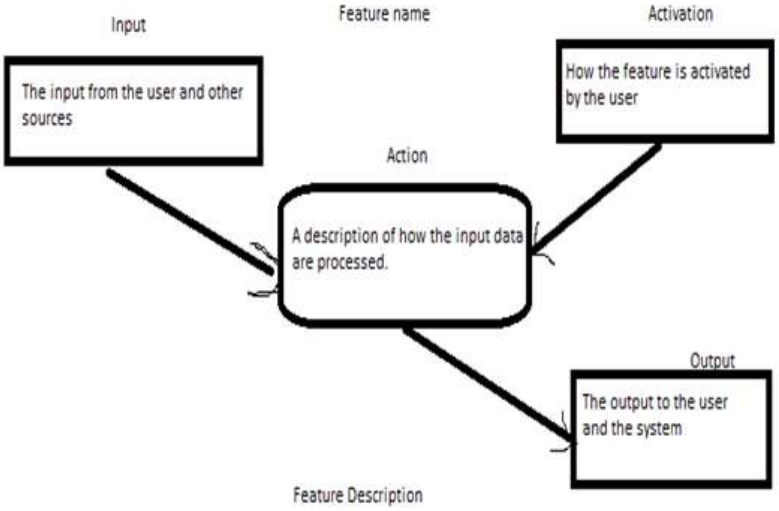


Figure 3.8 Feature design

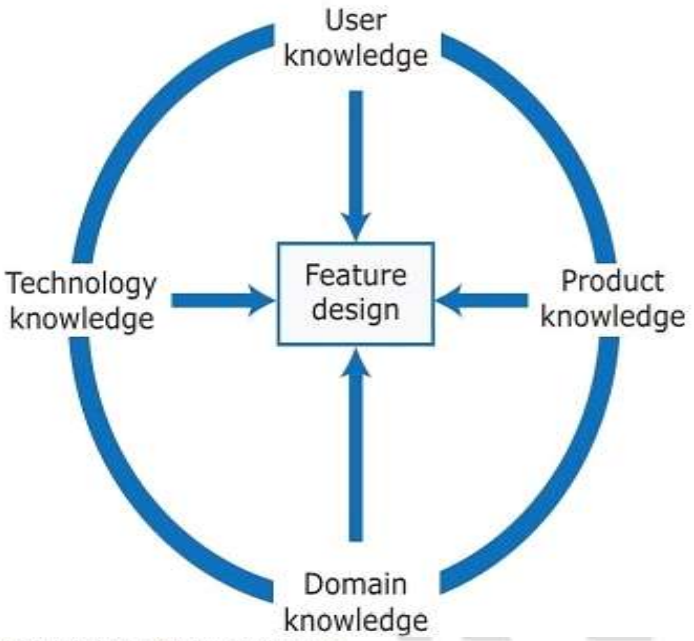
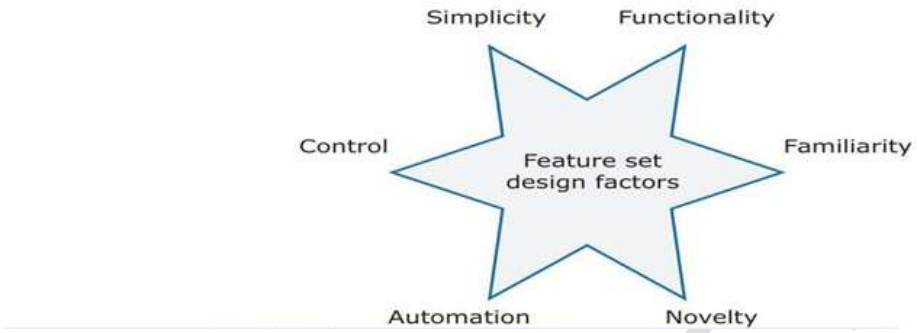


Table 3.8 Knowledge required for feature design

Knowledge	Description
User knowledge	You can use user scenarios and user stories to inform the team of what users want and how they might use the software features.
Product knowledge	You may have experience of existing products or decide to research what these products do as part of your development process. Sometimes your features have to replicate existing features in these products because they provide fundamental functionality that is always required.
Domain knowledge	This is knowledge of the domain or work area (e.g., finance, event booking) that your product aims to support. By understanding the domain, you can think of new innovative ways of helping users do what they want to do.
Technology knowledge	New products often emerge to take advantage of technological developments since their competitors were launched. If you understand the latest technology, you can design features to make use of it.

Figure 3.9 Factors in feature set design



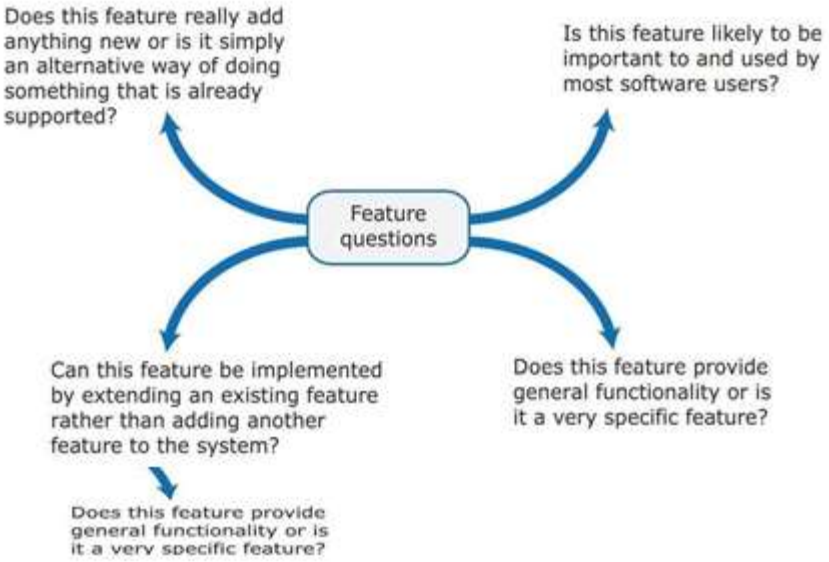
One problem that product developers should be aware of and try to avoid is "featurecreep."

- Feature creep the number of features in a product creeps potential uses of the product are envisaged.
- It adds to the complexity of a product, which means that you are likely to introduce bugs and security vulnerabilities into the software.
- It also usually makes the user interface more complex.

Feature creep happens for 3 reasons:

- Feature creep happens for 3 reasons:
- Product managers and marketing executives discuss the functionality they need with a range of different product users. Different users have slightly different needs or may do the same thing but in slightly different ways.
- Competitive products are introduced with slightly different functionality to your product. There is marketing pressure to include comparable functionality so that market share is not lost to these competitors. Thiscan lead to "feature wars," where competing products become more and more bloated as they replicate the features of their competitors.
- The product tries to support both experienced and inexperienced users. Easy ways of implementing common actions are added for inexperienced users and the more complex features to accomplish the same thing are retained because experienced users prefer to work that way.
- To avoid feature creep, the product manager and the development team should review all feature proposals and compare new proposals to features that have already been acceptedfor implementation.

Figure 3.10 Avoiding feature creep



Feature identification should be a team activity, and as features are identified, the team should discuss them and generate ideas about related features.

- Collaborative writing
- Blogs and web pages

Feature List

The output of the feature identification process should be a list of features that you use for designing and implementing your product.

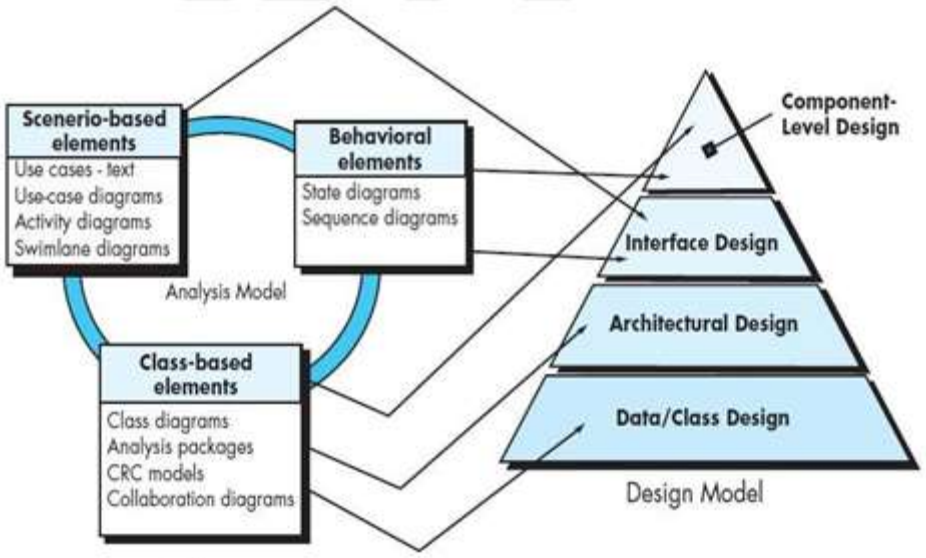
- Add detail when you are implementing the feature.
- You can describe a feature from one or more user stories.
- Scenarios and user stories should always be your starting point for identifying product features.

Design concepts - Design within the context of software engineering, Design Process, Design concepts, Design Model.

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering. Beginning once software requirements have been analyzed and modeled, software design is the last translating the requirements model into the design model.



Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated. The requirements model, manifested by scenario-based, class-based, and behavioral elements, feed the design task.

The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and provide the basis for the data design activity.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and patterns. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models and behavioral models serve as the basis for component design.

The importance of software design can be stated with a single word— *quality*. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow.

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software.

Quality Guidelines.

Consider the following guidelines:

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Assessing Design Quality—the Technical Review

During design, quality is assessed by conducting a series of technical reviews (TRs). A technical review is a meeting conducted by members of the software team. Usually two, three, or four people participate depending on the scope of the design information to be reviewed.

Quality Attributes. The FURPS quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ***Usability*** is assessed by considering human factors,overall aesthetics, consistency, and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured using processing speed, response time, resource consumption, throughput, and efficiency.
- ***Supportability*** combines extensibility, adaptability, and serviceability. These three attributes represent a more common term, *maintainability* —and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration),the ease with which a system can be installed, and the ease with which problems can be localized.

Common characteristics:

(1) A mechanism for the translation of the requirements model into a design representation,
(2) A notation for representing functional components and their interfaces,
(3) Heuristics for refinement and partitioning
(4) Guidelines for quality assessment.

DESIGN CONCEPTS

Abstraction

- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.
- A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

  An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

- A ***data abstraction***is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. A set of architectural patterns enables a software engineer to reuse design-level concepts.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design

Structural properties define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.".

- ***Extra-functional properties*** address **"**how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- ***Families of related systems*** **"**draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

- ***Structural models*** represent architecture as an organized collection of program components.
- ***Framework models*** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
- ***Dynamic models*** address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- ***Process models*** focus on the design of the business or technical process that the system must accommodate.
- **F*unctional models*** can be used to represent the functional hierarchy of a system.

Patterns

- *"A* pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns".
- A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.
- The intent of each design pattern is to provide a description that enables a designer to determine

- (1) Whether the pattern is applicable to the current work,
  (2) Whether the pattern can be reused (hence, saving design time), and
  (3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

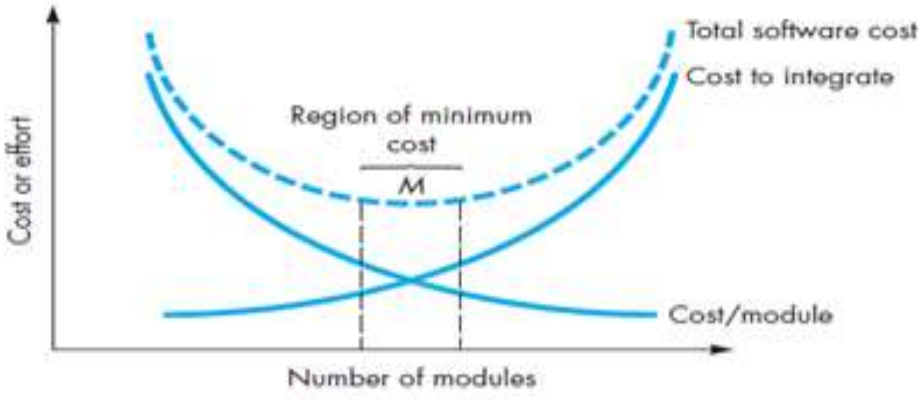
Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
A ***concern*** is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

- *Modularity* is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules that* are integrated to satisfy problem requirements.
- "Modularity is the single attribute of software that allows a program to be intellectually manageable".

Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number ofvariables, and overall complexity would make understanding close to impossible. In the Figure, the effort (cost) to develop an individual software module does decrease as the total number ofmodules increases.

- 

Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown inthe figure. There is a number, *M*, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict *M* with assurance.

Information Hiding

- The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others."
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

Functional Independence

- Functional independence is achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules.
- Functional independence is a key to good design, and design is the key to software quality.
- Independence is assessed using two qualitative criteria: **cohesion and coupling**. ***Cohesion*** is an indication of the relative functional strength of a module. ***Coupling*** is an indication of the relative interdependence among modules.
- A **cohesive** module performs a single task, requiring little interaction with other components in other parts of a program
- **Coupling** is an indication of interconnection among modules in a software structure.
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- High cohesion and low coupling make the module to be effectively design.

Refinement

- ***Stepwise refinement*** is a top-down design strategy.
- An application is developed by successively refining levels of procedural detail.

Aspects

An aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components.

Refactoring

- R*efactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.
- "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or in appropriate data structures, or any other design failure that can be corrected to yield a better design.

Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. OO design concepts such as classes and objects, inheritance, messages, and polymorphism

Design Classes

The analysis model defines a set of analysis classes . Five different types of design classes, each representing a different layer of the design architecture, can be developed

- ***User interface classes*** define all abstractions that are necessary for human-computer interaction (HCI) and often implement the HCI in the context of a metaphor.
- ***Business domain classes*** identify the attributes and services (methods) that are required to implement some element of the business domain that was defined by one or more analysis classes.
- ***Process classes*** implement lower-level business abstractions required to fully manage the business domain classes.
- ***Persistent classes*** represent data stores (e.g., a database) that will persist beyond the execution of the software.
- ***System classes*** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

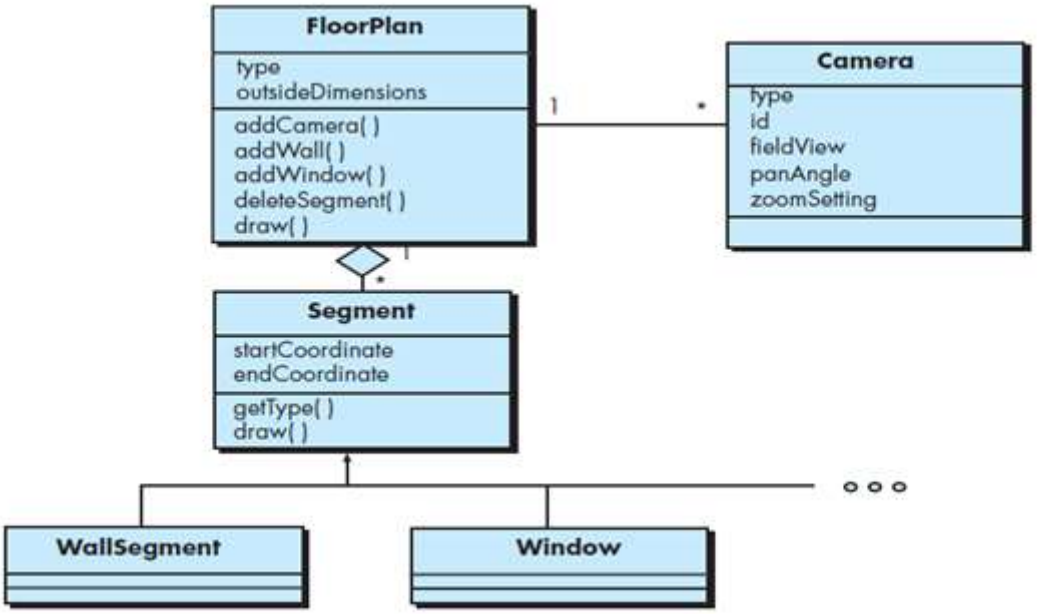
High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem

A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

- Refinement is actually **a process of *elaboration***. You begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- Abstraction and refinement are complementary concepts.
- Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details.
- Refinement helps you to reveal low-level details as design progresses.

Both concepts allow you to create a complete design model as the design evolves. should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* suggests that a method should only send messages to methods in neighboring classes.



Design class for Floor Plan and composite aggregation for the class

Dependency Inversion

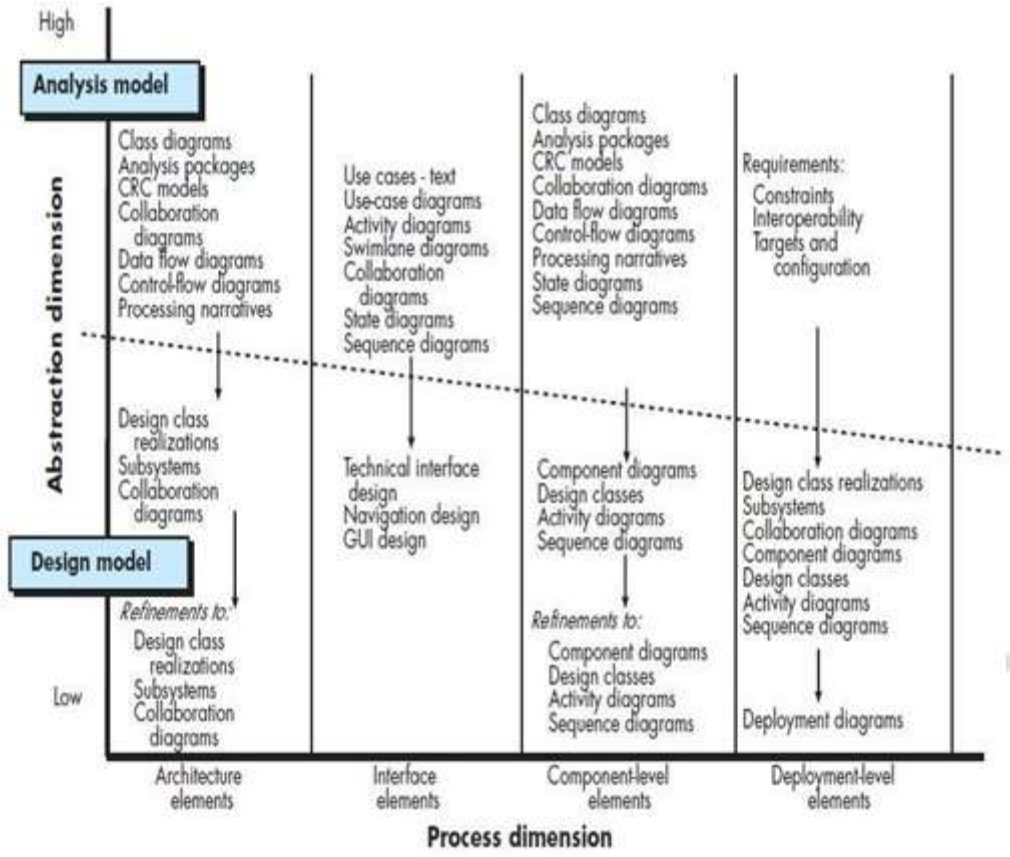
Dependency inversion principle which states: *High-level modules(classes) should not depend [directly] upon low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

Design for Test

There is an ongoing debate about whether software design or test case design should come first. Test-driven development (TDD) write tests before implementing any other code. They take to heart Tom Peters' credo, "Test fast, fail fast, and adjust fast." Testing guides their design as they implement in short, rapid-fi re "write test code—fail the test—write enough code to pass—then pass the test" cycles.

THE DESIGN MODEL

- The design model can be viewed in two different dimensions.
- The ***process dimension*** indicates the evolution of the design model as design tasks are executed as part of the software process.
- The ***abstraction dimension*** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. The dashed line indicates the boundary between the analysis and design models.
- However, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model is usually delayed until the design has been fully developed.



1 . Data Design Elements

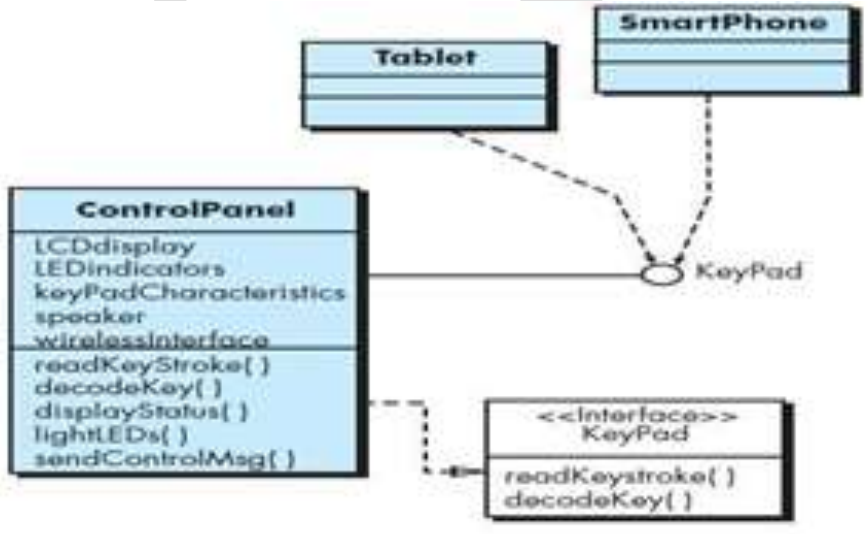
- Data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.
- The structure of data has always been an important part of software design. At the program-component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

2. Architectural Design Elements

The *architectural design* for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software. The architectural model is derived from three sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as use cases or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles and patterns.

Interface Design Elements

- The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan.
- The interface design elements for software depict information flows into and out of a system and how it is communicated among the components defined as part of the architecture.
- There are three important elements of interface design: (1) the user interface (UI), (2) external interfaces to other systems, devices, networks, or other producers or consumers of information, and (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.



Interface requirement for Control Panel

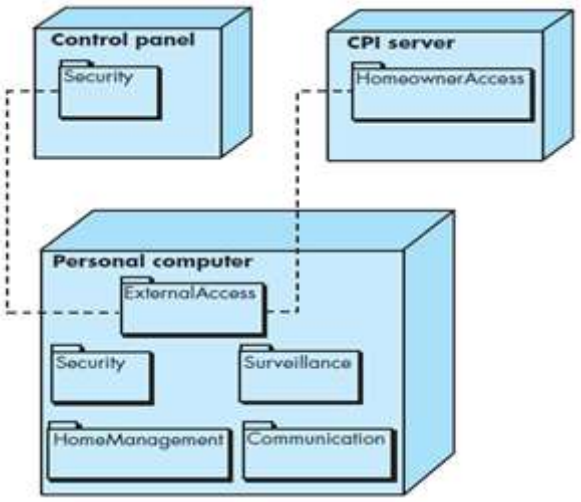
2. Component-Level Design Elements

- The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets, and every other detail associated with a room.
- The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

A UML component diagram

3. Deployment-Level Design Elements

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.
- For example, the elements of the *SafeHome* product are configured to operate within three primary computing environments—a homebasedPC, the *SafeHome* control panel, and a server housed at CPI Corp. (providing Internet-based access to the system). In addition, limited functionality may be provided with mobile platforms.
- During design, a UML deployment diagram is developed and then refined as shown in Figure. In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and functionality delivered by mobile platforms). The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features.
- In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome* system from an external source. Each subsystem would be elaborated to indicate the components that it implements.
- The diagram shown in Figure is in *descriptor form*. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further identified. It could be a Mac, a Windows-based PC, a Linux-box or a mobile platform with its associated operating system. These details are provided when the deployment diagram is revisited in *instance form* during the latter stages of design or as construction begins. Each instance of the deployment (a specific named hardware configuration) is identified.



A UML deployment diagram

ARCHITECTURAL DESIGN

Architectural Design - Software Architecture, Architectural Styles, Architectural considerations, Architectural Design

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

SOFTWARE ARCHITECTURE

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Identify three key reasons that software architecture is important:

- Software architecture provides a representation that facilitates communication among all stakeholders.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

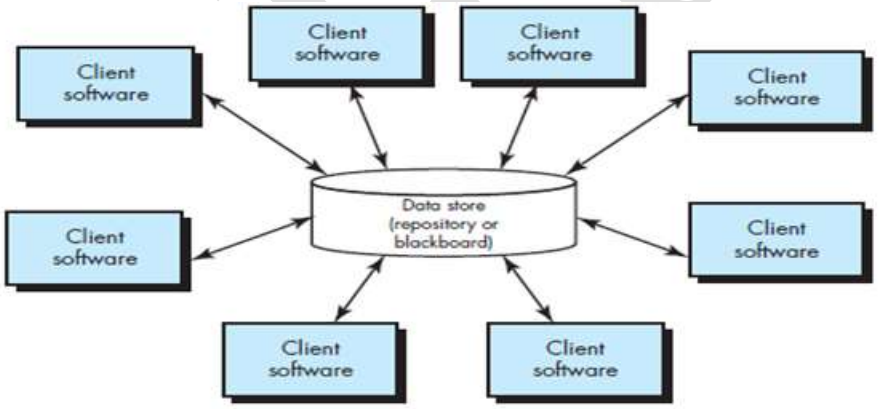
ARCHITECTURAL STYLES

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered, the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components.

Different Architectural Styles

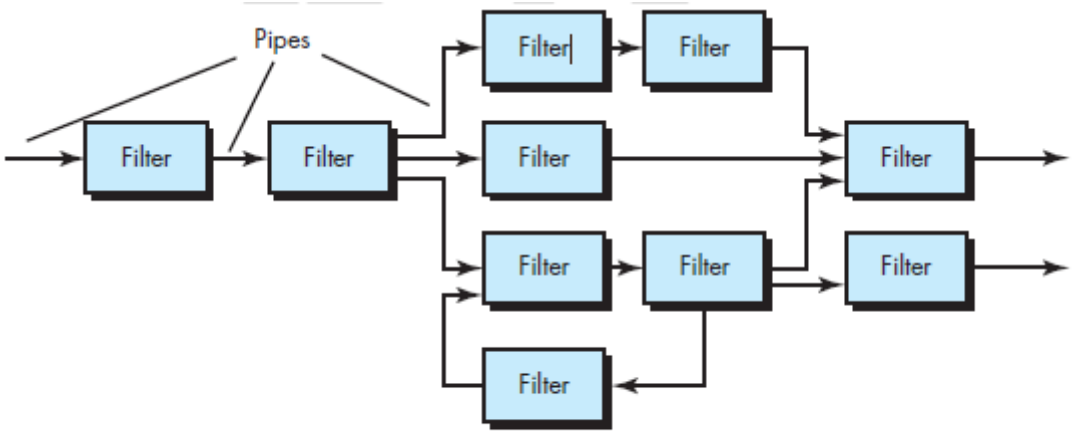
- **Data-Centered Architecture:** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote *integrability*. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.



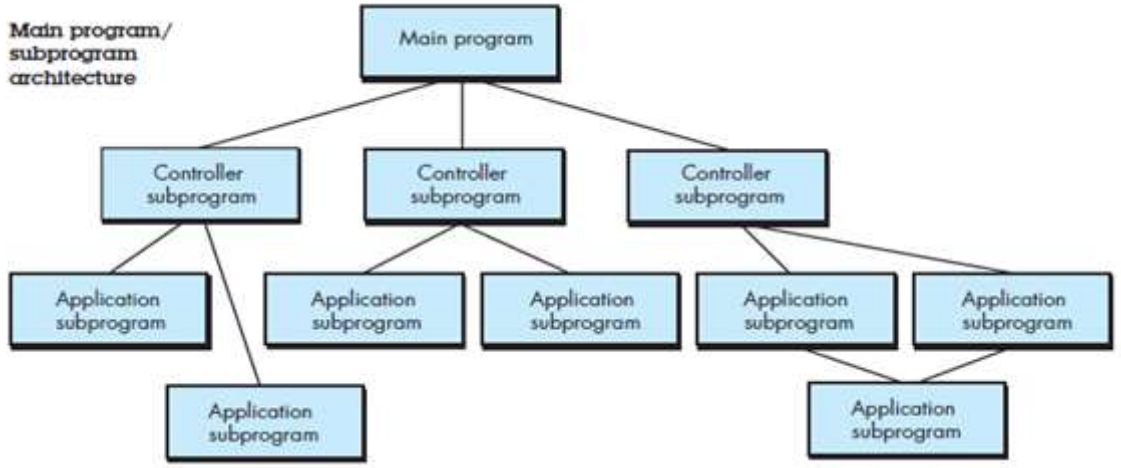
- **Data-Flow Architectures:** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A **pipe-and-filter** pattern has a set of components, called ***filters***, connected by ***pipes*** that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed ***batch sequential***. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.



Pipes and filters

- **Call and Return Architectures:** This architectural style enables to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category:
  - ***Main program/subprogram architectures***. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure above illustrates an architecture of this type.
  - ***Remote procedure call architectures***. The components of a main program/ subprogram architecture are distributed across multiple computers on a network.



- **Object-Oriented Architectures:** The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered Architectures:

- The basic structure of a layered architecture is illustrated. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.
-

ARCHITECTURAL CONSIDERATIONS

Buschmann and Henny suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made:

- **Economy** —many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or nonfunctional requirements (e.g., reusability when it serves no purpose). The best software is uncluttered and relies on abstraction to reduce unnecessary detail.
- **Visibility** —As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time. Poor visibility arises when important design and domain concepts are poorly communicated to those who must complete the design and implement the system.
- **Spacing—** Separation of concerns in a design without introducing hidden dependencies is a desirable design concept that is sometimes referred to as *spacing.* Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility.
- **Symmetry** —Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate. As an example of architectural symmetry, consider a *customer account* object whose life cycle is modeled directly by a software architecture that requires both *open ()* and *close()* methods. Architectural symmetry can be both structural and behavioral.

- **Emergence** —Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. The sequence and duration of the events that define the system's behavior is an emergent quality. It is very difficult to plan for every possible sequence of events. Instead the system architect should create a flexible system that accommodates this emergent behavior.

ARCHITECTURAL CONSIDERATIONS

Buschmann and Henny suggest several architectural considerations that can provide software engineers with guidance as architecture decisions are made:

- **Economy** —many software architectures suffer from unnecessary complexity driven by the inclusion of unnecessary features or nonfunctional requirements (e.g., reusability when it serves no purpose). The best software is uncluttered and relies on abstraction to reduce unnecessary detail.
- **Visibility** —As the design model is created, architectural decisions and the reasons for them should be obvious to software engineers who examine the model at a later time. Poor visibility arises when important design and domain concepts are poorly communicated to those who must complete the design and implement the system.
- **Spacing—** Separation of concerns in a design without introducing hidden dependencies is a desirable design concept that is sometimes referred to as *spacing.* Sufficient spacing leads to modular designs, but too much spacing leads to fragmentation and loss of visibility.
- **Symmetry** —Architectural symmetry implies that a system is consistent and balanced in its attributes. Symmetric designs are easier to understand, comprehend, and communicate. As an example of architectural symmetry, consider a *customer account* object whose life cycle is modeled directly by a software architecture that requires both *open ()* and *close()* methods. Architectural symmetry can be both structural and behavioral.
- **Emergence** —Emergent, self-organized behavior and control are often the key to creating scalable, efficient, and economic software architectures. For example, many real-time software applications are event driven. The sequence and duration of the events that define the system's behavior is an emergent quality. It is very difficult to plan for every possible sequence of events. Instead the system architect should create a flexible system that accommodates this emergent behavior.

ARCHITECTURAL DESIGN

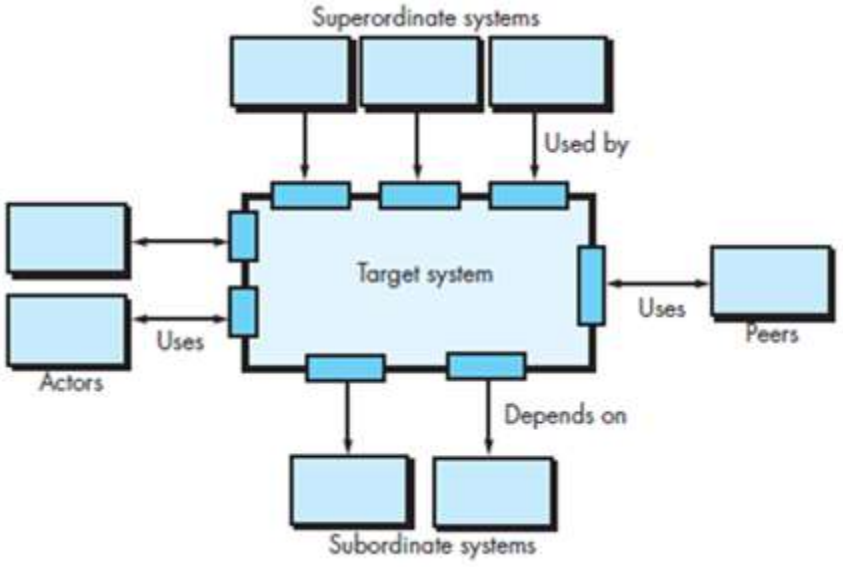
As architectural design begins, context must be established. To accomplish this, the external entities (e.g., other systems, devices, and people) that interact with the software and the nature of their interaction are described. This information can generally be acquired from the requirements model. Once context is modeled and all external software interfaces have been described to identify a set of architectural archetypes.

An ***archetype*** is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

Representing the System in Context

At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated.

Architectural context diagram
Source: Adapted from [Bos00].



Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as:

- *Superordinate systems* —those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems* —those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

- *Peer-level systems* —those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
- *Actors* —entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

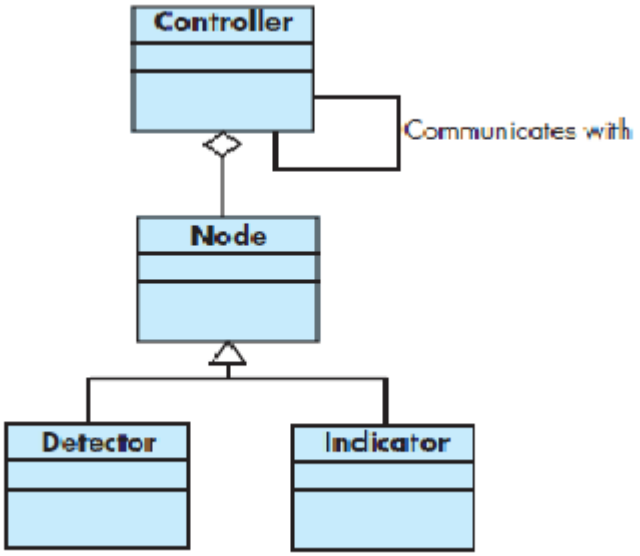
Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

Defining Archetypes

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *Safe Home* security function, you might define the following archetypes:

- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example, a node might be composed of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.
- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.



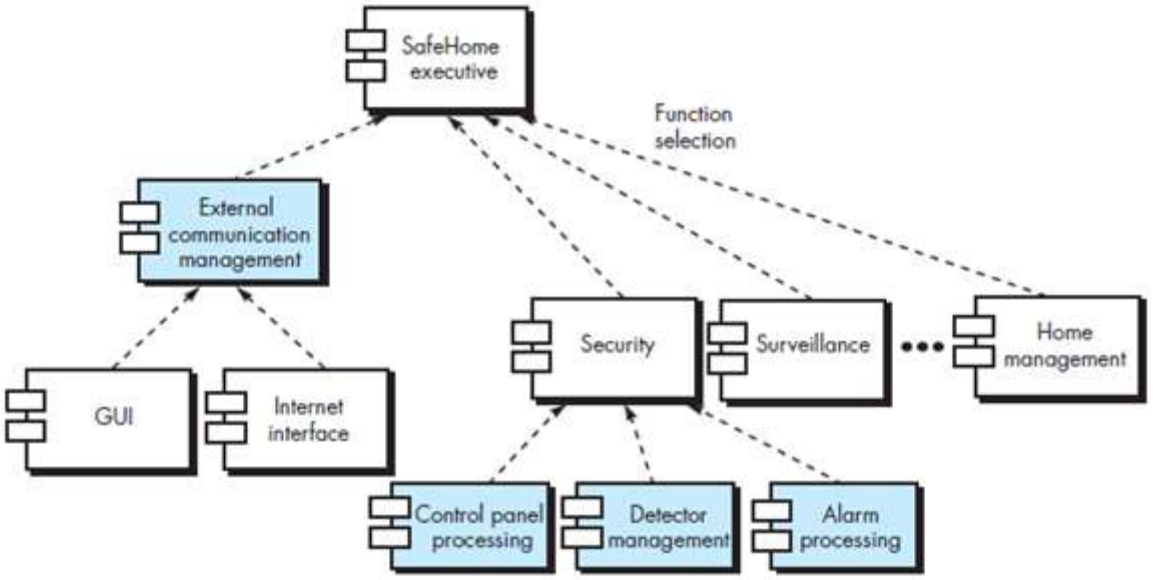
Each of these archetypes is depicted using UML notation as shown in Figure .**Detector** might be refined into a class hierarchy of sensors.

Refining the Architecture into Components

- As the software architecture is refined into components, the structure of the system begins to emerge. These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain.
- The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain.
- For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.
- The interfaces depicted in the architecture context diagram imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.
- Continuing the *Safe Home S*ecurity function example, you might define the set of top-level components that address the following functionality:
- *External communication management* —coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- *Control panel processing* —manages all control panel functionality.
- *Detector management* —coordinates access to all detectors attached to the system.
- *Alarm processing* —verifies and acts on all alarm conditions.

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure. Transactions are acquired by *external communication management* as they move in from components that process the *SafeHome* GUI and the Internet interface. This information is managed by a *SafeHome* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

Overall architectural structure for *SafeHome* with top-level components



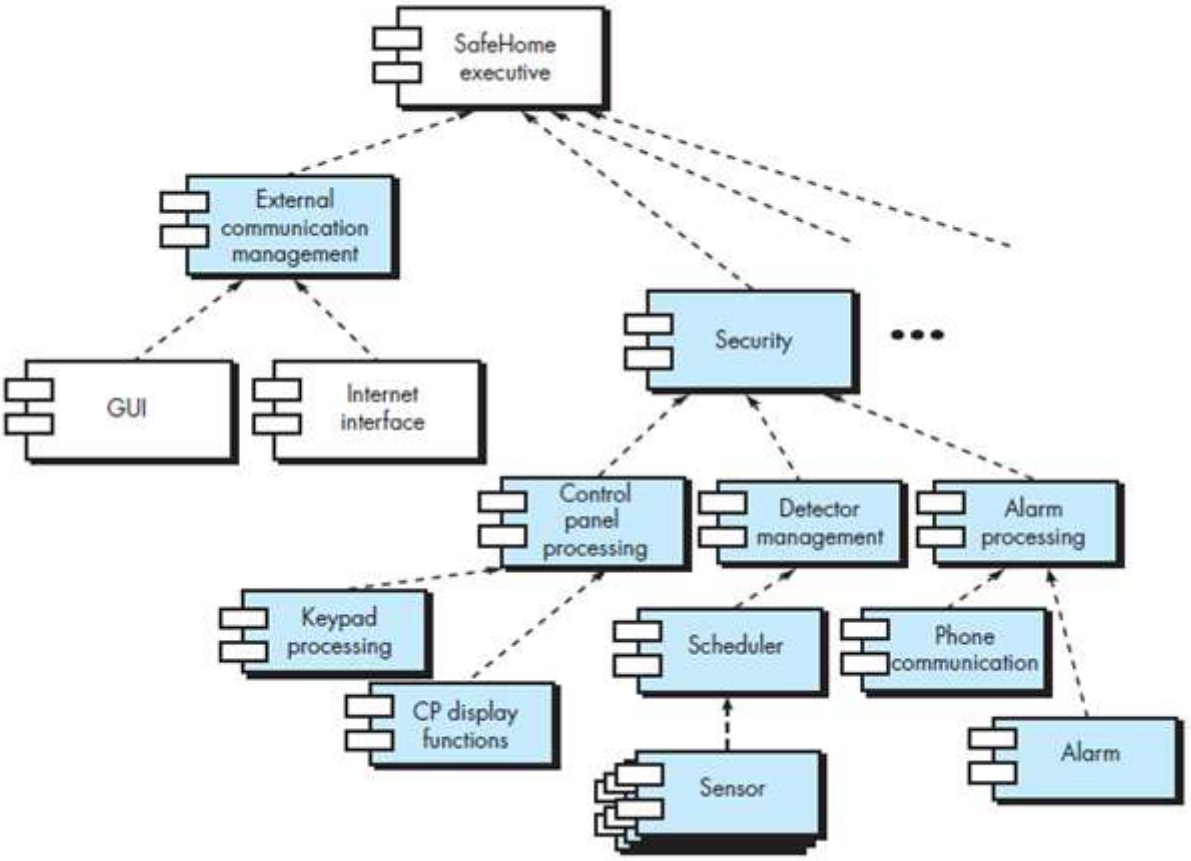
Describing Instantiations of the System

- The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented; archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this we mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure above are elaborated to show additional detail. For example, the *detector management* component interacts with a *scheduler* infrastructure component that implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure below:

An instantiation of the security function with component elaboration



- WebApps are client-server applications typically structured using multilayered architectures, including a user interface or view layer, a controller layer which directs the flow of information to and from the client browser based on a set of business rules, and a content or model layer that may also contain the business rules for the WebApp.
- The user interface for a WebApp is designed around the characteristics of the web browser running on the client machine (usually a personal computer or mobile device). Data layers reside on a server. Business rules can be implemented using a server-based scripting language such as PHP or a client-based scripting language such as JavaScript. An architect will examine requirements for security and usability to determine which features should be allocated to the client or server.
- The architectural design of a WebApp is also influenced by the structure (linear or nonlinear) of the content that needs to be accessed by the client. The architectural components (Web pages) of a WebApp are designed to allow control to be passed to other system components, allowing very flexible navigation structures. The physical location of media and other content resources also influences the architectural choices made by software engineers.

Architectural Design for Mobile Apps

- Mobile apps are typically structured using multilayered architectures, including a user interface layer, a business layer, and a data layer. With mobile apps you have the choice of building a thin Web-based client or a rich client. With a thin client, only the user interface resides on the mobile device, whereas the business and data layers reside on a server. With a rich client all three layers may reside on the mobile device itself.
- Mobile devices differ from one another in terms of their physical characteristics (e.g., screen sizes, input devices), software (e.g., operating systems, language support), and hardware (e.g., memory, network connections). Each of these attributes shapes the direction of the architectural alternatives that can be selected.
- A number of considerations that can influence the architectural design of a mobile app: (1) the type of web client (thin or rich) to be built, (2) the categories of devices (e.g., smart phones, tablets) that are supported, (3) the degree of connectivity (occasional or persistent) required, (4) the bandwidth required, (5) the constraints imposed by the mobile platform, (6) the degree to which reuse and maintainability are important, and (7) device resource constraints (e.g., battery life, memory size, processor speed).

COMPONENT LEVEL DESIGN

Component

A component is a modular building block for computer software. The OMG Unified Modeling Language Specification defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation andexposes a set of interfaces."

Components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, and people) that exist outside the boundaries of the software.

The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

Three important views of what a component is and how it is used as designmodeling proceeds.

An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, we begin with the analysis model and elaborate analysis classes(for components that relate to the problem domain) and infrastructure classes (for components (for components that provide support services for the problem domain).

The Traditional View

A traditional component called a *module,* resides within the software architecture and serves one of three important roles: (1) a *control component* that coordinates the invocation of all other problem domain components, (2) a *problem domain component* that implements a complete or partial function that is required by the customer, or (3) an *infrastructure component* that is responsible for functions that support the processing required in the problem domain.

A Process-Related View

Over the past three decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. A catalog of proven design or code-level components is made available to you as design work proceeds. As the software architecture is developed, we choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to you.

DESIGN CLASS BASED COMPONENTS

Basic design principles

The Open-Closed Principle (OCP). "A module [component] should be open for extension but closed for modification"

- Should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself.
- To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.
- One way to accomplish OCP for the **Detector** class is illustrated in Figure .The sensor interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the **Detector** class(component). The OCP is preserved.

The Liskov Substitution Principle (LSP).

- "*Subclasses should be substitutable for their base classes"*.
- This design principle suggests that a component that uses a base class should continueto function properly if a class derived from the base class is passed to the component instead
- In the context, a "contract" is a *pre-condition* that must be true before the component uses a base class and a *post-condition* that should be true after the component uses a base class. When you create derived classes, be sure they conform to the pre- and post-conditions.

Dependency Inversion Principle (DIP).

- "*Depend on abstractions. Do not dependon concretions"*.
- Abstractions are the place where a design can be extended without great complication. The more a component depends on other concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP).

- *"Many client-specific interfaces are better than one general purpose interface"*.
- There are many instances in which multiple client components use the operations provided by a server class.
- Should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of specialized interfaces.

The Release Reuse Equivalency Principle (REP).

- *"The granule of reuse is the granule of release"*
- When classes or components are designed for reuse, an implicit contract is established between the developer of the reusable entity and the people who will use it.
- The developer commits to establish a release control system that supports and maintains older versions of the entity while theusers slowly upgrade to the most current version.

The Common Closure Principle (CCP).

- *"Classes that change together belong together."*
- Classes should be packaged cohesively.
- That is, when classes are packaged as part of a design, they should address the same functional or behavioural area.
- When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads tomore effective change control and release management

The Common Reuse Principle (CRP).

- *"Classes that aren't reused together should not be grouped together"*
- When one or more classes with a package changes, the release number of the package changes.
- All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operated without incident.
- If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing.
- For this reason, only classes that are reused together should be included within a package.

Component-Level Design Guidelines

These guidelinesapply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design.

Suggests the following guidelines:

Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated aspart of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model. For example, the class

We can choose to use stereotypes to help identify the nature of components at the detailed design level. For example, <<infrastructure>> might be used to identify an infrastructure component, <<database>> could be used to identify a database that services one or more design classes or the entire system; <<table>> can be used to identify a table within a database.

Interfaces. Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OPC).

Dependencies and Inheritance. For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).

Cohesion

Implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

Functional. Exhibited primarily by operations, this level of cohesion occurs when a module performs one and only one computation and then returns aresult.

Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Classes and components that exhibit **functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.**

Coupling

- As the amount of communication and collaboration increases (i.e., as the degree of "connectedness" between classes increases), the complexity of the system also increases. And as complexity increases, the difficulty of implementing, testing, and maintaining software grows.
- ***Coupling*** is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.
- ***Content coupling*** occurs when one component "surreptitiously modifies data that is internal to another component" .This violates information hiding—a basic design concept.

- ***Control coupling*** occurs when operation *A()* invokes operation *B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.
- ***External coupling*** occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, tele-communication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.
- Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to reduce coupling whenever possible

CONDUCTING COMPONENT LEVEL DESIGN

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. Classes and components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. Messages that are passed between objects within a system.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. ".

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation.

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudo code or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept.

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or sub function. The next iteration does little more than expand the operation name.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioural representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behaviour of the system and the more localized behaviour of individual analysis classes. During component-level design, it is sometimes necessary to model the behaviour of a design class.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state (mode of behavior) of the object. To understand the dynamic behavior of an object, you should examine all use cases that are relevant to the design class throughout its life.

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions (oftenrepresented as subsystems) are represented within the context of the computing environment that will house them.

During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components. However, components generally are not represented individually within a component diagram. In some cases, deployment diagrams are elaborated into instance form at this time. This means that the specific hard- ware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated.

Step 7. Refactor every component-level design representation and always con- sider alternatives. Design is an iterative process. The first component-level model we create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model. It is essential to refactor as design work is conducted.

COMPONENT LEVEL DESIGN FOR WEB APPLICATIONS

WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

Content Design at the Component Level

Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user. The formality of content design at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity (of the WebApp, content objects, and their interrelation-ships) grows, it may be necessary to organize content in a way that allows easier reference and design manipulation. In addition, if content is highly dynamic (e.g., the content for an online auction site), it becomes important to establish a clear structural model that incorporates content components.

Functional Design at the Component Level

WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure consistency. We beginby considering both the requirements model and the initial information architecture and then examining how functionality affects the user's interaction with the application, the information that is presented, and the user tasks that are conducted.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A ***functional architecture*** is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other

Design Document Template

Contents

CASE STUDY

Ariane 5 launch accident

This case study describes the accident that occurred on the initial launch of the Ariane 5 rocket, a launcher developed by the European Space Agency. The rocket exploded shortly after take-off and the subsequent enquiry showed that this was due to a fault in the software in the inertial navigation system.

In June 1996, the then new Arianne 5 rocket was launched on its maiden flight. It carried a payload of scientific satellites. Ariane 5 was commercially very significant for the European Space Agency as it could carry a much heavier payload than the Ariane 4 series of launchers. Thirty seven seconds into the flight, software in the inertial navigation system, whose software was reused from Ariane 4, shut down causing incorrect signals to be sent to the engines. These swivelled in such a way that uncontrollable stresses were placed on the rocket and it started to break up. Ground controllers initiated self-destruct and the rocket and payload was destroyed.

A subsequent enquiry showed that the cause of the failure was that the software in the inertial reference system shut itself down because of an unhandled numeric exception (integer overflow). There was a backup software system but this was not diverse so it failed in the same way.

The Ariane 5 launcher failure

While developing the Ariane 5 space launcher, the designers decided to reuse the inertial reference software that had performed successfully in the Ariane 4 launcher. The inertial reference software maintains the stability of the rocket. The designers decided to reuse this without change (as you would do with components), although it included additional functionality that was not required in Ariane 5.

In the first launch of Ariane 5, the inertial navigation software failed, and the rocket could not be controlled. The rocket and its payload were destroyed. The cause of the problem was an unhandled exception when a conversion of a fixed-point number to an integer resulted in a numeric overflow. This caused the runtime system to shut down the inertial reference system, and launcher stability could not be maintained. The fault had never occurred in Ariane 4 because it had less powerful engines and the value that was converted could not be large enough for the conversion to overflow.

This illustrates an important problem with software reuse. Software may be based on assumptions about the context where the system will be used, and these assumptions may not be valid in a different situation.

More information about this failure is available at: http://software-engineering-book.com/case-studies/ariane5/

Module 3 Notes

Module 3

Review Techniques - Cost impact of Software Defects, Code review and statistical analysis. Informal Review, Formal Technical Reviews, Post-mortem evaluations. Software testing strategies - Unit Testing, Integration Testing, Validation testing, System testing, Debugging, White box testing, Path testing, Control Structure testing, Black box testing, Testing Documentation and Help facilities. Test automation, Test-driven development, Security testing. Overview of DevOps and Code Management - Code management, DevOps automation, Continuous Integration, Delivery, and Deployment (CI/CD/CD). Software Evolution - Evolution processes, Software maintenance.

Review Techniques

- Software reviews are a "filter" for the software process. That is, reviews are applied at various points during software engineering and serve to uncover errors and defects that can then be removed.
- Software reviews "purify" software engineering work products, including requirements and design models, code, and testing data. Many different types of reviews can be conducted as part of software engineering.
- Focus on *technical or peer reviews,* exemplified by *casual reviews, walkthroughs,* and *inspections.* A technical review (TR) is the most effective filter from a quality control standpoint. Conducted by software engineers (and others) for software engineers, the TR is an effective means for uncovering errors and improving software quality.

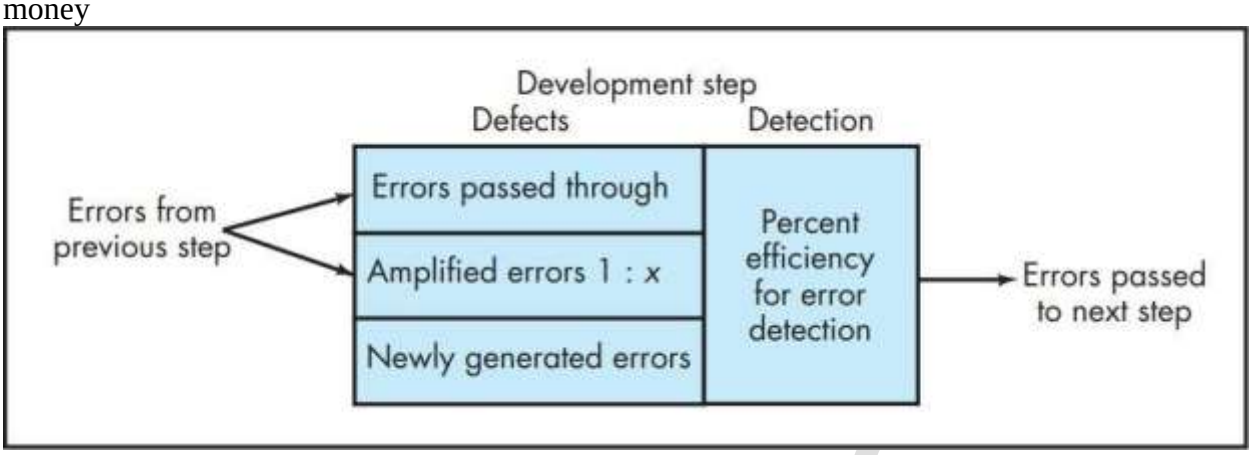
COST IMPACT OF SOFTWARE DEFECTS

- The primary objective of technical reviews is to find errors during the process so that they do not become defects after release of the software. The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.
- *The primary objective of an FORMAL TECHNICAL REVIEW (FTR) is to find errors before they are passed on to another software engineering activity or released to the end user.*
- Within the context of the software process, the terms defect and fault are synonymous. Both imply a quality problem that is discovered after the software has been released to end users (or to another framework activity in the software process).
- The primary objective of technical reviews is to find errors during the process so that they do not become defects after release of the software.
- The obvious benefit of technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.
  Review techniques have been shown to be up to 75 percent effective in uncovering design flaws. detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent activities in the software process.

DEFECT AMPLIFICATION AND REMOVAL

• A defect amplification model can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process.

• To conduct reviews, you must expend time and effort, and your development organization must spend money



REVIEW METRICS AND THEIR USE

• Technical reviews are one of many actions that are required as part of good software engineering practice.

• Each action requires dedicated human effort

• **Preparation effort, Ep**—the effort (in person-hours) required to review a work product prior to the actual review meeting

• **Assessment effort, Ea**— the effort (in person-hours) that is expended during the actual review

• **Rework effort, Er** — the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review

• **Work product size, WPS**—a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)

• **ANALYZING MATRICES**

• **Minor errors found, Errminor**—the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)

• **Major errors found, Errmajor**—the number of errors found that can be categorized as major (requiring more than some prespecifi ed effort to correct

• The total review effort and the total number of errors discovered are defi ned as: Ereview = Ep + Ea + Er Errtot = Errminor + Errmajor

• Error density represents the errors found per unit of work product reviewed.

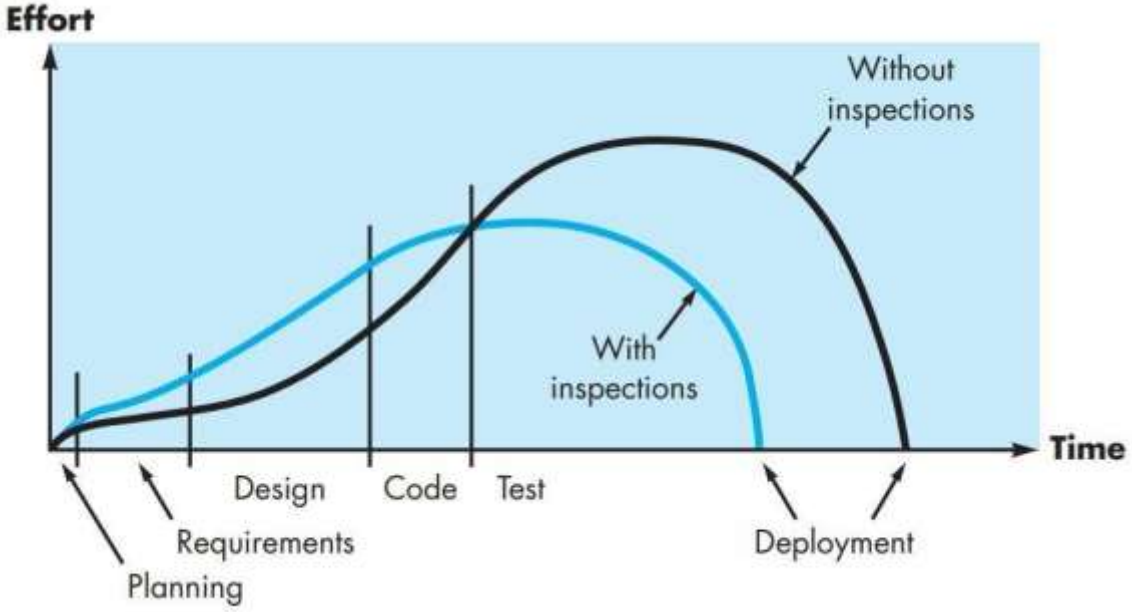
Error density = Errtot/ WPS

Cost-Effectiveness of Reviews

- It is difficult to measure the cost-effectiveness of any technical review in real time.
- A software engineering organization can assess the effectiveness of reviews and their cost benefit only after reviews have been completed, review metrics have been collected, average data have been computed, and then the downstream quality of the software is measured
- Effort saved per error = Etesting - Ereviews

Effort expended with and without reviews

- Effort expended when reviews are used does increase early in the development of a software increment.
- testing and corrective effort is reduced.
- The deployment date for development with reviews is sooner than the deployment date without reviews.
- Reviews don't take time, they save it.



REVIEWS : A FORMALITY SPECTRUM

- Technical reviews should be applied with a level of formality that is appropriate for the product to be built, the project time line, and the people who are doing the work.
- The formality of a review increases when
- Distinct roles are explicitly defined for the reviewers,
- There is a sufficient amount of planning and preparation for the review,
- A distinct structure for the review (including tasks and internal work products) is defined,
- A set of specific tasks would be conducted based on an agenda that was developed before the review occurred.
  - The results of the review would be formally recorded, and the team would decide on the status of the work product based on the outcome of the review.
  - Members of the review team might also verify that the corrections made were done properly

Informal Reviews

- Informal reviews include a simple desk check of a software engineering work product with a colleague, a casual meeting (involving more than two people) for the purpose of reviewing a work product
- A simple *desk check* or a *casual meeting* conducted with a colleague is a review. However, because there is no advance planning or preparation, no agenda or meeting structure, and no follow-up on the errors that are uncovered, the effectiveness of such reviews is considerably lower than more formal approaches. But a simple desk check can and does uncover errors that might otherwise propagate further into the software process.

REVIEW TECHNIQUES

FORMAL TECHNICAL REVIEWS

A *formal technical review* (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are: (1) to uncover errors in function, logic, or implementation for any representation of the software; (2) to verify that the software under review meets its requirements; (3) to ensure that the software has been represented according to predefined standards; (4) to achieve software that is developed in a uniform manner; and (5) to make projects more manageable. In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software.

The FTR is actually a class of reviews that includes ***walkthroughs*** **and** ***inspections***. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough are presented as a representative formal technical review. If you have interest in software inspections, as well as additional information on walkthroughs

Code Walkthrough

- We present the code and accompanying documentation to the review team, and the team comments on their correctness.
- During walkthrough, we lead and control the discussion. The atmosphere is informal and the focus of attention is on the code, not the coder.
- Although Supervisory personnel may be present, walkthrough has no influence on the performance appraisal, consistent with the general intent of testing, finding faults, not fixing them.

Code Inspection

- Similar to Code walkthrough, but is more formal. In an inspection, review team checks the code and documentation against a prepared list of concerns.
- For eg: the team may examine the definition and use of data type and structures to see if their use is consistent with the design and with standards and procedures. The team can review algorithms and computations for their correctness and efficiency. Interfaces also checked. The team may estimate the code's performance characteristics in terms of memory usage or processing speed.

Inspecting code usually involves several steps.

- First , the team may meet as a group for overview of the code and a description of the inspection goals.
- Then team members prepare individually for a second group meeting. Each inspector studies the code and its related documents, noting faults found. Finally in a group meeting, team members report what they have found, recording additional faults discovered in the process of discussing individuals findings. Sometimes faults discovered by an individual are considered to be false positives": items a that seem to be faults but in-fact were not considered by the group to be true problems.

The Review Meeting

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- The focus of the FTR is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component).
- The individual who has developed the work product—the producer— informs the project leader that the work product is complete and that a review is required.
- The project leader contacts a review leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.
- Each reviewer is expected to spend between one and two hours reviewing the product, making notes.
- The review meeting is attended by the review leader, all reviewers and the producer.
- One of the reviewers takes on the role of a recorder, that is, the individual who records (in writing) all important issues raised during the review.
- The FTR begins with an introduction of the agenda and a brief introduction by the producer.
- The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each.
- At the end of the review, all attendees of the FTR must decide whether to:

  A) accept the product without further modification,

  B) reject the product due to severe errors (once corrected, another review must be performed), or

C) Accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required).

- After the decision is made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.
- Review Reporting and Record Keeping
- During the FTR, a reviewer (the recorder) actively records all issues that have been raised.
- These are summarized at the end of the review meeting, and a review issues list is produced.
- In addition, a formal technical review summary report is completed. A review summary report answers three questions:
- What was reviewed?
- Who reviewed it?
- What were the findings and conclusions?

The review summary report is a single-page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties. The review issues list serves two purposes:
(1) to identify problem areas within the product and
(2) to serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

Review Guidelines

The following represents a minimum set of guidelines for formal technical reviews:

1. Review the product, not the producer.
2. Set an agenda and maintain it.
3. Limit debate and rebuttal.
4. Enunciate problem areas, but don't attempt to solve every problem noted.
5. Take written notes.
6. Limit the number of participants and insist upon advance preparation.
7. Develop a checklist for each product that is likely to be reviewed.
8. Allocate resources and schedule time for FTRs.
9. Conduct meaningful training for all reviewers.
10. Review your early reviews

POST-MORTEM EVALUATIONS

- ***Post-mortem evaluation*** (PME) as a mechanism to determine what went right and what went wrong when software engineering process and practice are applied in a specific project.
- Unlike an FTR that focuses on a specific work product, a PME examines the entire software project, focusing on both " ***excellences*** (that is, achievements and positive experiences) and ***challenges*** (problems and negative experiences)"
- PME is attended by members of the software team and stakeholders. The intent is to identify excellences and challenges and to extract lessons learned from both.
- To determine whether quality control activities are working, a set of metrics should be collected. Review metrics focus on the effort required to conduct the review and the types and severity of errors uncovered during the review. Once metrics data are collected, they can be used to assess the efficacy of the reviews you do conduct. Industry data indicates that reviews provide a significant return on investment.

SOFTWARE TESTING STRATEGIES

A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. A strategy for software testing is developed by the project manager, software engineers, and testing specialists. Testing begins "in the small" and progresses "to the large." By this we mean that early testing focuses on a single component or on a small group of related components and applies tests to uncover errors in the data and processing logic that have been encapsulated by the component(s). After components are tested they must be integrated until the complete system is constructed. At this point, a series of high-order tests are executed to uncover errors in meeting customer requirements. As errors are uncovered, they must be diagnosed and corrected using a process that is called debugging.

A number of **software testing strategies** have been proposed ,all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews . By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). ***Verification*** refers to the set of tasks that ensure that software correctly implements a specific function. ***Validation*** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way: Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

Software Testing Strategy—The Big Picture

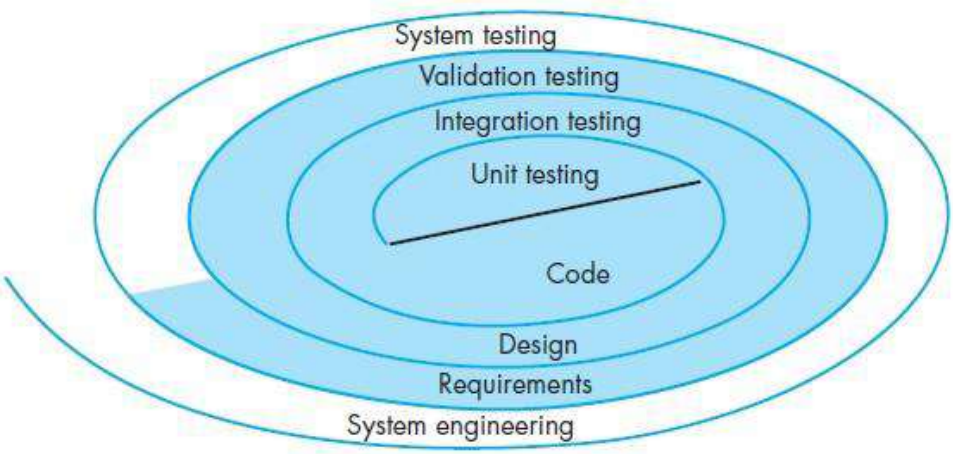
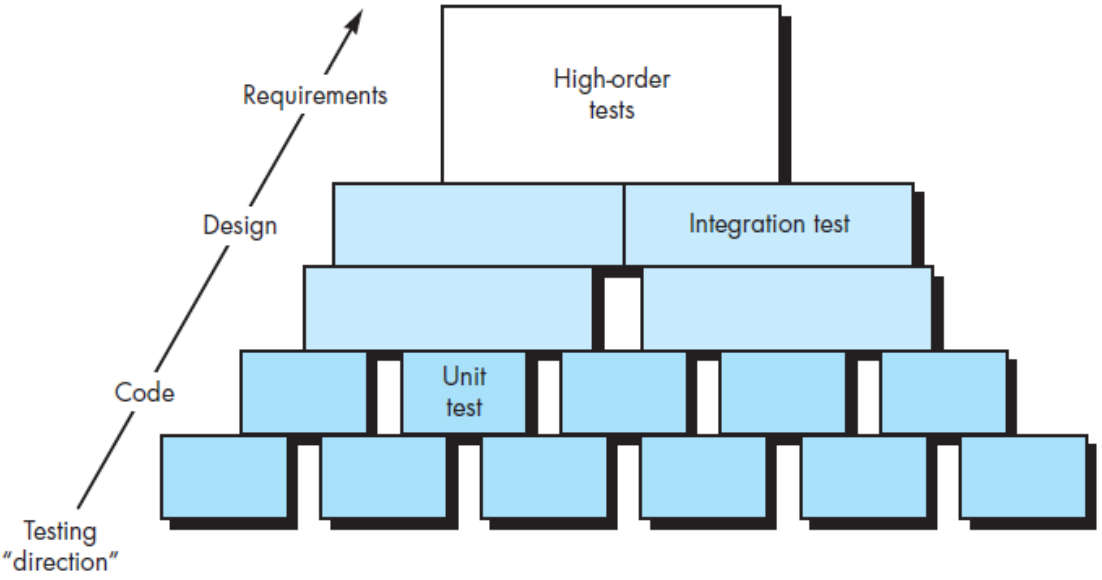


Figure Testing Strategy

A strategy for software testing may also be viewed in the context of the spiral (Figure).

- *Unit testing* begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.
- Testing progresses by moving outward along the spiral to *integration testing*, where the focus is on design and the construction of the software architecture.
- Taking another turn outward on the spiral, you encounter *validation testing*, where requirements established as part of requirements modeling are validated against the software that has been constructed.
- Finally, you arrive at *system testing*, where the software and other system elements are tested as a whole. To test computer software, you spiral out along streamlines that broaden the scope of testing with each turn.
- Initially, tests focus each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing***. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.
- Next, components must be assembled or integrated to form the complete software package. *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.
- After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated. *Validation testing* provides final assurance that software meets all functional, behavioral, and performance requirements.

Figure : Software Testing steps



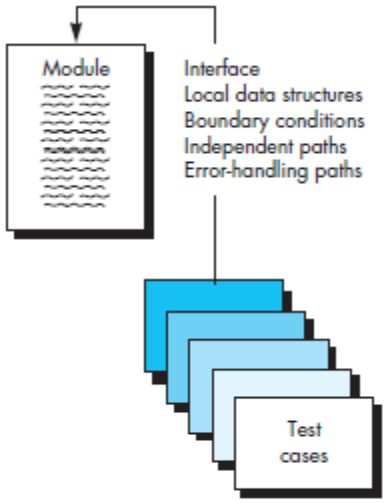
- The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases).
- *System testing* verifies that all elements mesh properly and that overall system function/performance is achieved.

Unit Testing

- *Unit testing* focuses verification effort on the smallest unit of software design—the software component or module.
- Using the component-level design description, important control paths are tested to uncover errors within the boundary of the module.
- The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

Unit Test Considerations.

- Unit tests are illustrated schematically in Figure.



- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error handling paths are tested.
- Data flow across a component interface is tested before any other testing is initiated.
- If data do not enter and exit properly, all other tests are moot.
- In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.
- Selective testing of execution paths is an essential task during the unit test.
- Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.
- Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n* th element of an *n* -dimensional array is processed, when the *i*th repetition of a loop with *I* passes is invoked, when the maximum or minimum allowable value is encountered.

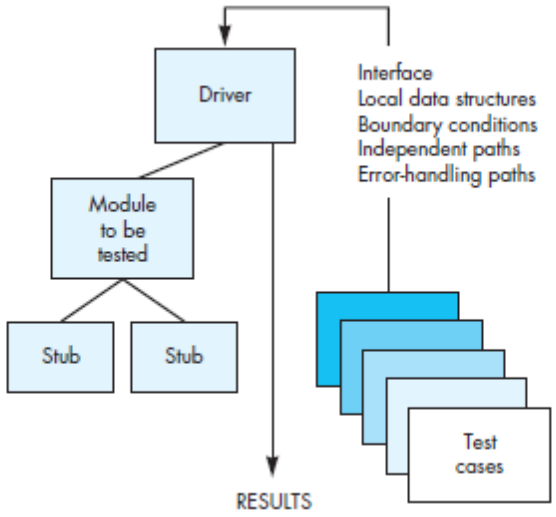
- Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.
- A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur -*antibugging*.
- Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible,

  (2) error noted does not correspond to error encountered,
- error condition causes system intervention prior to error handling, exception-condition processing is incorrect, or error description does not provide enough information to assist in the location of the cause of the error.

Unit-Test Procedures.

- Unit testing is normally considered as an adjunct tothe coding step. The design of unit tests can occur before coding begins or aftersource code has been generated.
- A review of design information provides guidancefor establishing test cases that are likely to uncover errors in each of thecategories .
- Each test case should be coupled with a set of expected results.

- Because a component is not a stand-alone program, driver and/or stub softwaremust often be developed for each unit test.
- The unit test environment isillustrated in Figure .



- In most applications a *driver* is nothing more than a"main program" that accepts test-case data, passes such data to the component(to be tested), and prints relevant results.

- *Stubs* serve to replace modules that aresubordinate (invoked by) the component to be tested.

- A stub or "dummy subprogram"uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoingtesting.
- Drivers and stubs represent testing "overhead." That is, both are software thatmust be coded (formal design is not commonly applied) but that is not deliveredwith the final software product.
- If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequatelyunit tested with "simple" overhead software.
- In such cases, complete testing canbe postponed until the integration test step (where drivers or stubs are also used).

Integration Testing

- Components must be assembled or integrated to form the complete software package. where the focus is on design and the construction of the software architecture.
- *Integration testing* addresses the issues associated with the dual problems of verification and program construction. Testcase design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths.
- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.
- To construct the program using a "**big bang**" approach. All components are combined in advance and the entire program is tested as a whole. Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- **Incremental integration** is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Top-Down Integration.

- *Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth first or breadth-first manner. Referring to Figure below ,*depth-first integration* integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.
- For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated.
- Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.
- The integration process is performed in a series of five steps:
- The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
- Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
- Tests are conducted as each component is integrated.
- On completion of each set of tests, another stub is replaced with the real component.
- Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

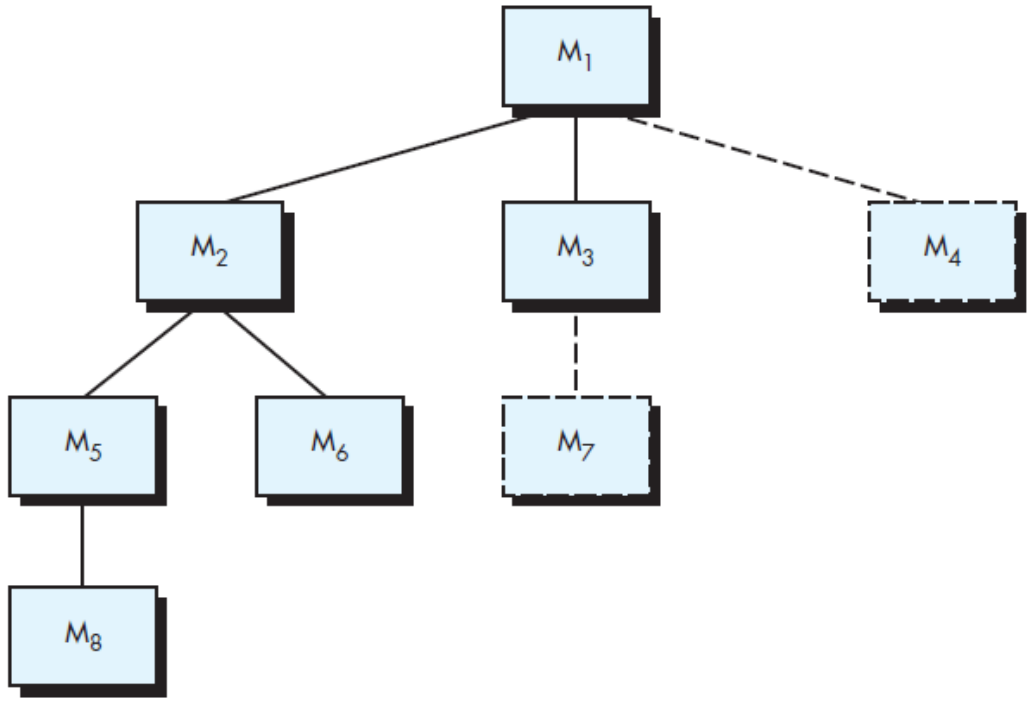


Figure : Top down integration

- The process continues from step 2 until the entire program structure is built. The top-down integration strategy verifies major control or decision points early in the test process. In a "well- factored" program

structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. Early demonstration of functional capability is a confidence builder for all stakeholders.

Bottom-Up Integration.

Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
2. A *driver* (a control program for testing) is written to coordinate test-case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

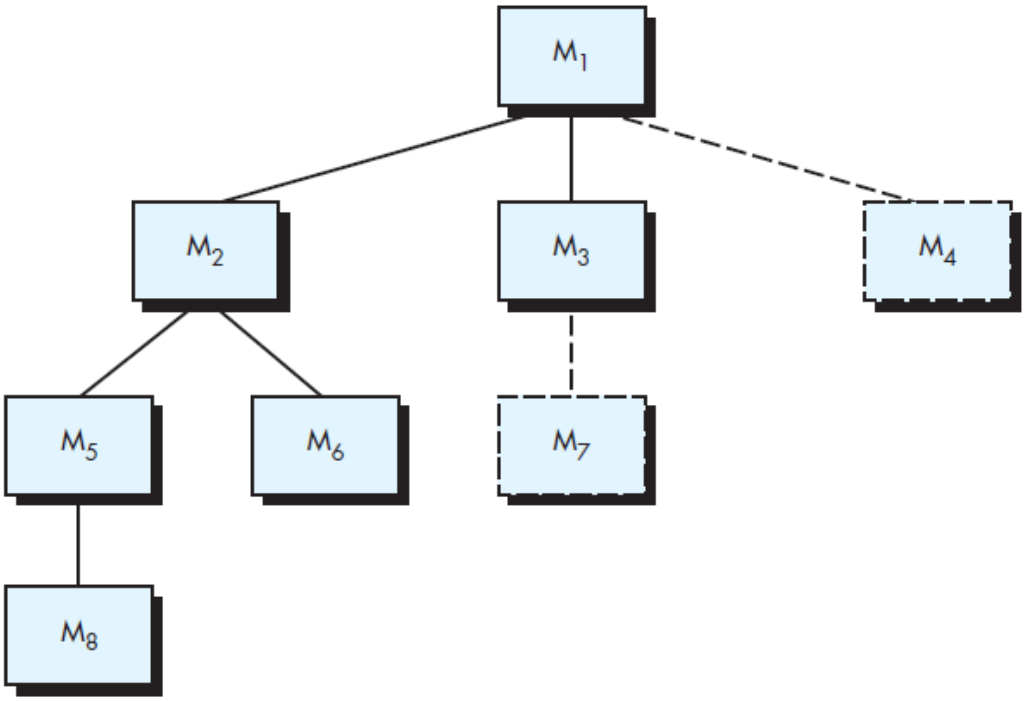


Figure : Bottom up approach

Integration follows the pattern illustrated in Figure . Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M*a* . Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a* . Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb.
Both M*a* and M*b* will ultimately be integrated with component M*c* , and so forth. As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression Testing.

- Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. Side effects associated with these changes may cause problems with functions that previously worked flawlessly.
- Regression testing refers to a type of software testing that is used to verify any modification or update in a software without affecting the overall working functionality of the said software**.**
- Test cases are re-executed to check the previous functionality of the application is working fine, and the new changes have not produced any bugs.
- In the context of an integration test strategy, ***regression testing*** is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools.
- ***Capture/playback tools*** enable the software engineer to capture test cases and results for subsequent playback and comparison.

- The ***regression test suite*** (the subset of tests to be executed) contains three different classes of CST test cases:
- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

- As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

Smoke Testing.

- ***Smoke testing*** is an integration testing approach that is commonly used when product software is developed.
- Smoke testing is a process where the software build is deployed to quality assurance environment and is verified to ensure the stability of the application. Smoke Testing is also known as ***Confidence Testing* or *Build Verification Testing*.**
- **Smoke Testing** is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing. It consists of a minimal set of tests run on each build to test software functionalities.
- It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

- **Smoke Testing** is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing.
- It consists of a minimal set of tests run on each build to test software functionalities. Smoke testing is also known as "**Build Verification Testing" or "Confidence Testing**."
- In essence, the smoke-testing approach encompasses the following activities:

  1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show-stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.
- The smoke test should exercise the entire system from end to end.
- Smoke testing provides a number of benefits when it is applied on complex, time-critical software projects:

- ***Integration risk is minimized***. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- ***The quality of the end product is improved***. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- ***Error diagnosis and correction are simplified***. Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- ***Progress is easier to assess***. With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

TEST STRATEGIES FOR WEBAPPS

- The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems.
- The following steps summarize the approach:
- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Each functional component is unit tested.
- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for errors.

TEST STRATEGIES FOR MOBILEAPPS

- The strategy for testing mobile applications adopts the basic principles for all software testing. However, the unique nature of MobileApps demands the consideration of a number of specialized testing approaches:
- *User-experience testing.* Users are involved early in the development process to ensure that the MobileApp lives up to the usability and accessibility expectations of the stakeholders on all supported devices.
- *Device compatibility testing.* Testers verify that the MobileApp works correctly on all required hardware and software combinations.
- *Performance testing.* Testers check nonfunctional requirements unique to mobile devices (e.g., download times, processor speed, storage capacity, power availability).

- *Connectivity testing.* Testers ensure that the MobileApp can access any needed networks or Web services and can tolerate weak or interrupted network access.
- *Security testing* . Testers ensure that the MobileApp does not compromise the privacy or security requirements of its users.
- *Testing-in-the-wild* . The app is tested under realistic conditions on actual user devices in a variety of networking environments around the globe.
- Certification testing. Testers ensure that the MobileApp meets the standards established by the app stores that will distribute it.

C .VALIDATION TESTING

- The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.
- Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.
- Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between different software categories disappears. Testing focuses on user- visible actions and user-recognizable output from the system.

C1 :Validation-Test Criteria

- Software validation is achieved through a series of tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted, and a test procedure define specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).
- If a deviation from specification is uncovered, a *deficiency list* is created. A method for resolving deficiencies (acceptable to stakeholders) must be established.

C2 :Alpha and Beta Testing

- When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.
- Conducted by the end user rather than software engineers, an acceptance test can range from an informal “test drive” to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.
- The ***alpha test*** is conducted at the developer’s site by a representative group of end users. The software is used in a natural setting with the developer “looking over the shoulder” of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.
- The ***beta test*** is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a “live” application of the software in an environment that cannot be controlled by the developer.
- The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.
- A variation on beta testing, called ***customer acceptance testing***, is sometimes performed when custom

software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer.

D: SYSTEM TESTING

D.1: Recovery Testing

- Many computer-based systems must recover from faults and resume processing with little or no downtime.
- In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.
- ***Recovery testing*** is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

D.2: Security Testing

- Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain.
- Security testing is an integral part of software testing, which is used to discover the weaknesses, risks, or threats in the software application and also help us to stop the nasty attack from the outsiders and make sure the security of our software applications.
- The primary objective of security testing is to find all the potential ambiguities and vulnerabilities of the application so that the software does not stop working. If we perform security testing, then it helps us to identify all the possible security threats and also help the programmer to fix those errors.
- It is a testing procedure, which is used to define that the data will be safe and also continue the working process of the software.

D.3 Stress Testing

- **Stress Testing** is a type of software testing that verifies stability & reliability of software application.
- The goal of Stress testing is measuring software on its robustness and error handling capabilities under extremely heavy load conditions and ensuring that software doesn't crash under crunch situations. It even tests beyond normal operating points and evaluates how software works under extreme conditions. It is also known as ***Endurance Testing, fatigue testing*** or ***Torture Testing***.
- The stress testing includes the **testing beyond standard operational size**, repeatedly to a **breaking point**, to get the outputs.
- It highlights the error handling and robustness under a heavy load instead of correct behavior under regular conditions.
- In other words, we can say that **Stress testing** is used to verify the constancy and dependability of the system and also make sure that the system would not crash under disaster circumstances.
- *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate 10 interrupts per second, when one or two is the average rate, (2)input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially,

the tester attempts to break the program.

- A variation of stress testing is a technique called *sensitivity testing*. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

D.4 Performance Testing

- ✓ Performance testing is a non-functional software testing technique that determines how the stability, speed, scalability, and responsiveness of an application holds up under a given workload .For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process.

  Even at the unit level, the performance of an individual module may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

  Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis

D.5 Deployment Testing

 Software must execute on a variety of platforms and under more than one operating system environment.
 Deployment testing, sometimes called *configuration testing*, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g.,“installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

E :THE ART OF DEBUGGING

☐ *Debugging* occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.

 Although debugging can and should be an orderly process, it is still very much an art.

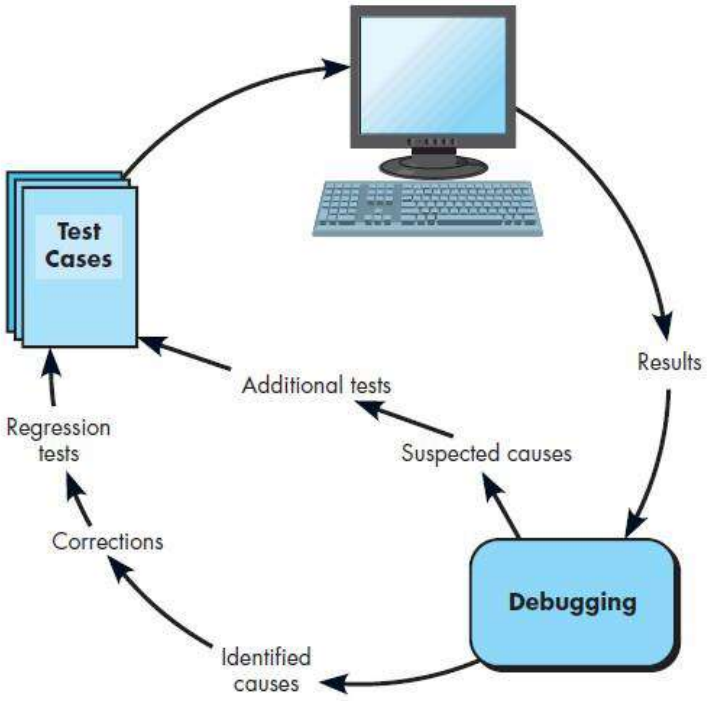


Figure 22.7 Debugging process

E.1 The Debugging Process

Debugging is not testing but often occurs as a consequence of testing, the debugging process begins with the execution of a test case.

Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction. The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components (Chapter 12) exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing Problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real- time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to fi nd the cause also increases. Often, pressure forces a software developer to fi x one error and at the same time introduce two more.

E2 Debugging Strategies

- Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.
- Debugging has one overriding objective—
- to find and correct the cause of a software error or defect.
- The objective is realized by a combination of systematic evaluation, intuition, and luck. In general, three debugging strategies have been proposed**: brute force, backtracking, and cause elimination**. Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Debugging Tactics.

- The ***brute force*** **category** of debugging is probably the most common and least efficient method for isolating the cause of a software error.
- This is the foremost common technique of debugging however is that the least economical method. during this approach, the program is loaded with print statements to print the intermediate values with the hope that a number of the written values can facilitate to spot the statement in error. This approach becomes a lot of systematic with the utilization of a symbolic program (also known as a source code debugger), as a result of values of various variables will be simply checked and breakpoints and watch-points can be easily set to check the values of variables effortlessly.

- ***Backtracking*** is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.
- The third approach to debugging**—** ***cause elimination***—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.
- A "cause hypothesis"is devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Automated Debugging.

- Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted.
- Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation."
- A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

SOFTWARE TESTING FUNDAMENTALS

- The goal of testing is to find errors, and a good test is one that has a high probability of finding an error.

The tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

- Testability
- Operability
- Observability.
- Controllability.
- Decomposability.
- Simplicity
- Stability.
- Understandability.

Test Characteristics

- A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).
- A good test should be "best of breed" .In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that has the highest likelihood of uncovering a whole class of errors.
- A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately

INTERNAL AND EXTERNAL VIEWS OF TESTING

Any engineered product can be tested in one of two ways:

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. :The first test approach takes an external view and is **called black-box testing.**

(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The second requires an internal view and is termed **white-box testing.**

A: WHITE-BOX TESTING

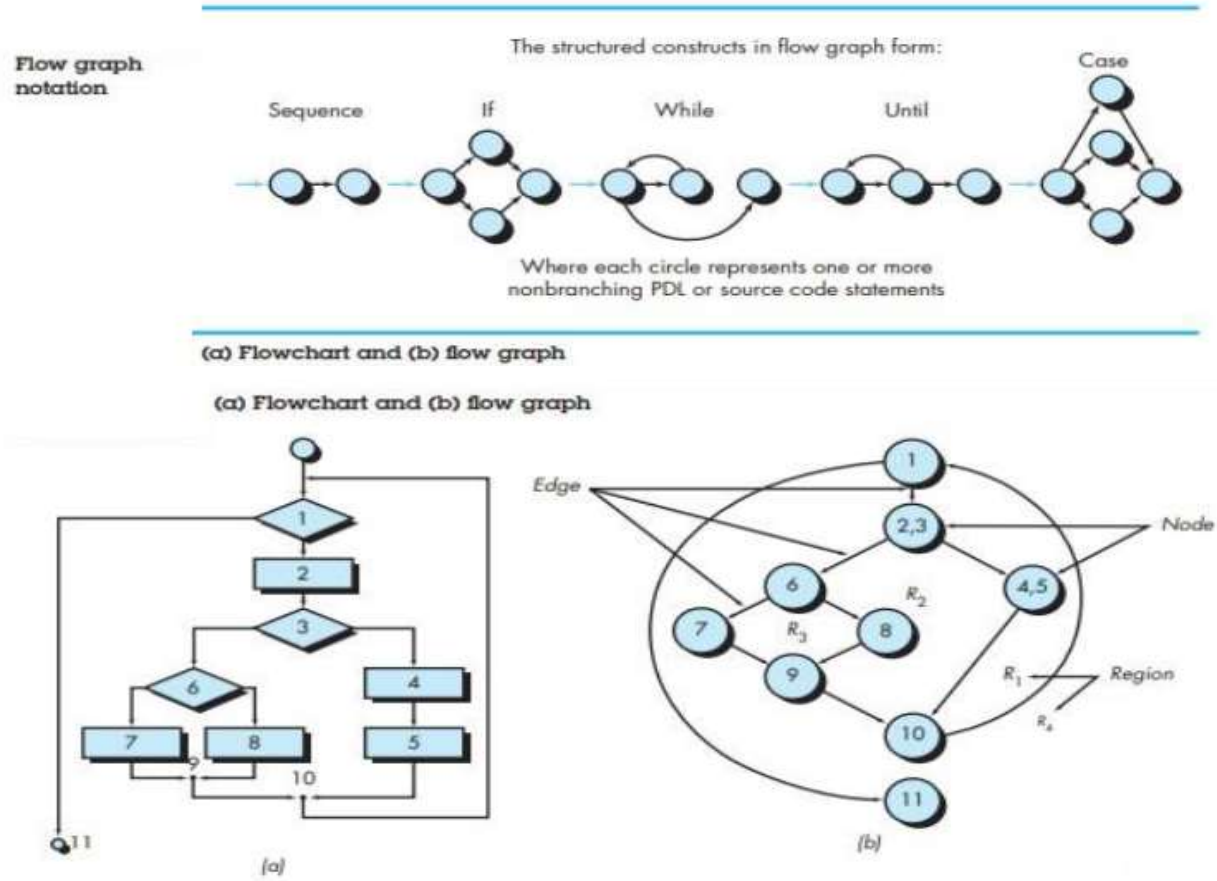
White-box is testing, sometimes called glass-box testing or structural testing, is a test- case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, we can derive test cases **that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.**

A1: BASIS PATH TESTING

Basis path testing is a white-box testing technique which enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

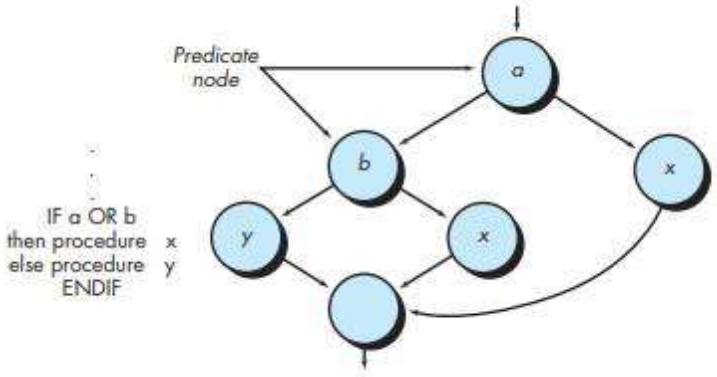
A.1.1 Flow Graph Notation

The flow graph depicts logical control flow using the notation illustrated in Figure



- To illustrate the use of a flow graph, consider the procedural design representation in Figure 23.2a . Here, a flowchart is used to depict program control structure.
- Figure 23.2bmaps the flowchart into a corresponding flow
- Referring to Figure 23.2b, each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node. The arrows on the w graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct)
- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.



A.1.2 Independent Program Paths

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 23.2 b is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Paths 1 through 4 constitute a basis set for the flow graph in Figure 23.2b . That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

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

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

1. **The number of regions of the flow graph corresponds to the cyclomatic complexity.**

1. **Cyclomatic complexity V(G) for a flow graph G is defined as V(G) =E-N+ 2 where E is the number of flow graph edges and N is the number of flow graph nodes.**
2. **Cyclomatic complexity V(G) for a flow graph G is also defined as V(G) =P+1**

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

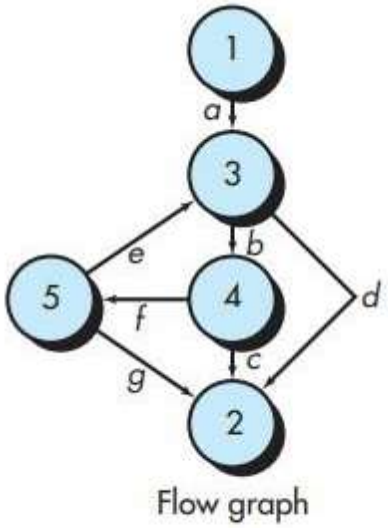
Referring once more to the flow graph in Figure 23.2b , the cyclomatic complexity can be computed using each of the algorithms just noted:
1. The flow graph has **four** regions.
2. V(G) = 11 edges - 9 nodes + 2 = 4.
3. V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 23.2bis 4.
More important, the value for V(G) provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements

A.1.3 **Graph Matrices**

A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing.
A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix Figure below.
Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.
To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

• The probability that a link (edge) will be executed.
• The processing time expended during traversal of a link
• The memory required during traversal of a link
• The resources required during traversal of a link.



Flow graph

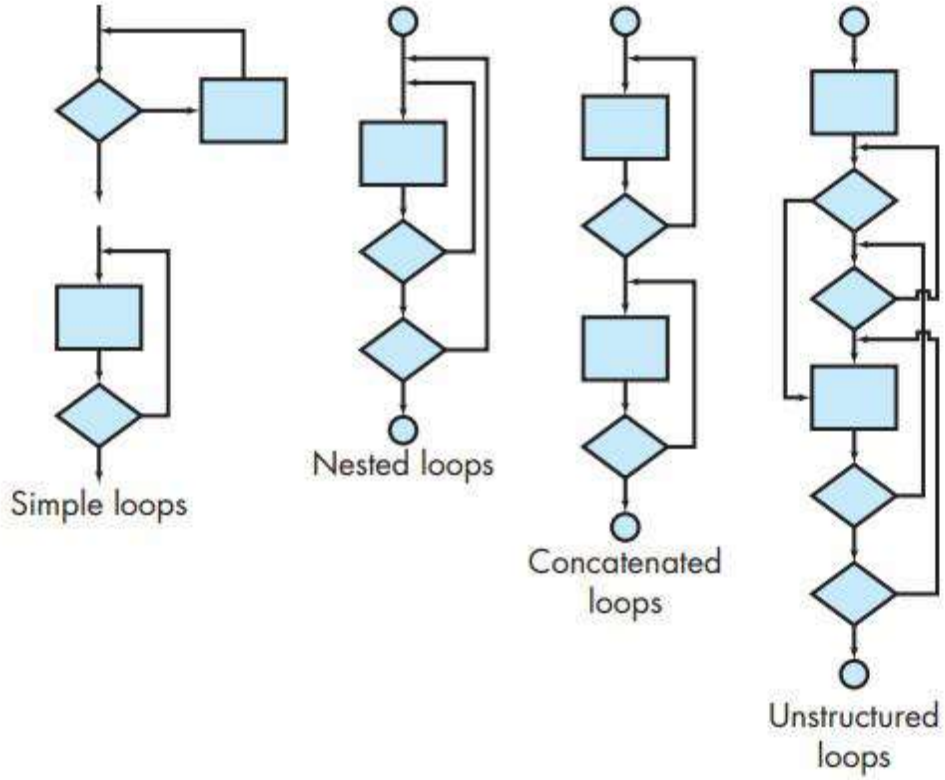
Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

2. CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for control structure testing. Condition testing is a test-case design method that exercises the logical conditions contained in a program module. Data flow testing selects test paths of a program according to the locations of definitions and uses of variables in the program.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops



Simple Loops. The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.
1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m < n.
5. n - 1, n, n + 1 passes through the loop.

Nested Loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
   Continue until all loops have been tested.

Concatenated Loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.
Unstructured Loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

3. BLACK-BOX TESTING

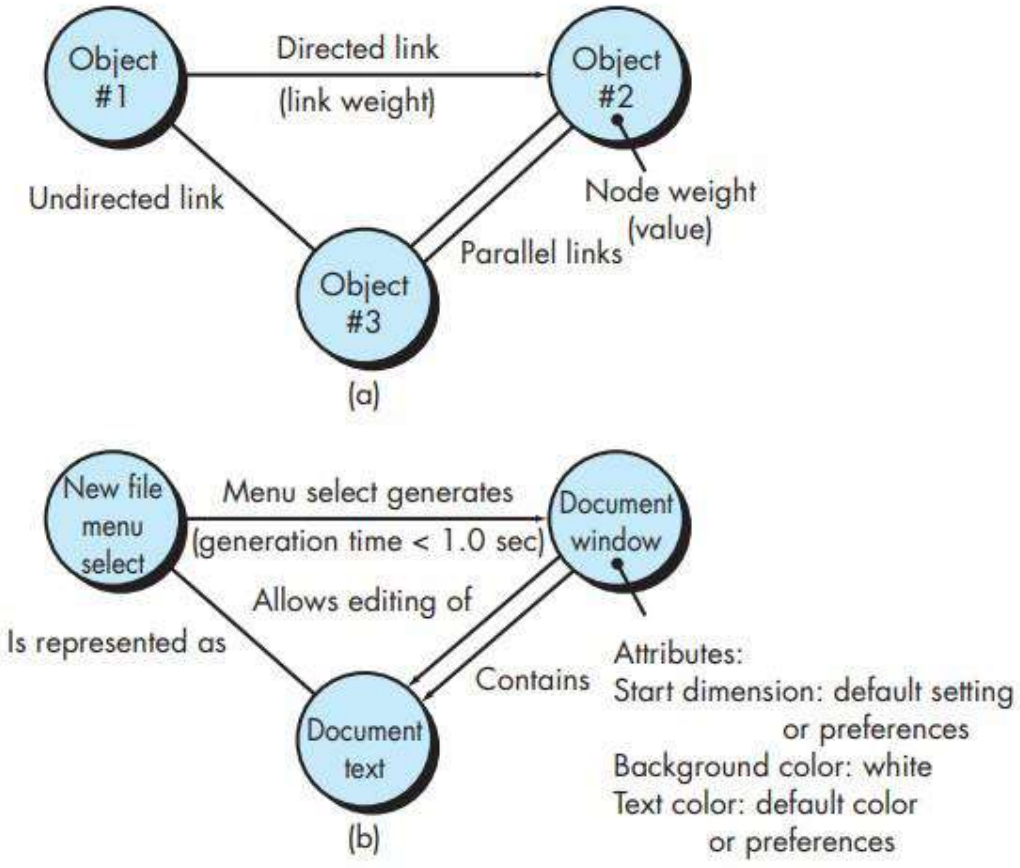
Black-box testing, also called behavioral testing or functional testing focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program

- Black-box testing attempts to find errors in the following categories:

(1) incorrect or missing functions,
(2) interface errors,
(3) errors in data structures or external database access,
(4) behavior or performance errors, and
(5) initialization and termination errors.

- Black-box testing is focused on the information domain. Black-box tests are designed to validate functional requirements without regard to the internal workings of a program.
- Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage.

Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function.
Boundary value analysis probes the program's ability to handle data at the limits of acceptability.
Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters.
Model-based testing uses elements of the requirements model to test the behavior of an application.
Graph-Based Testing Methods
The first step in black-box testing is to understand the objects 5 that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another" . Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.



To accomplish these steps, you begin by creating a graph—a collection of nodes that represent objects, links that represent the relationships between objects, node weights that describe the properties of a node (e.g., a specific data value or state behavior), and link weights that describe some characteristic of a link.

The symbolic representation of a graph is shown in Figure a . Nodes are represented as circles connected by links that take a number of different forms. A **directed link** (represented by an arrow) indicates that a relationship moves in only one direction. A **bidirectional link,** also
called a symmetric link, implies that the relationship applies in both directions. **Parallel links** are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 23.8b) where Object #1 = newFile (menu selection) Object #2 = documentWindow Object #3 = document Text Referring to the figure, a menu select on newFile generates a document window. The node weight of documentWindow provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the newFile menu selection and documentText, and parallel links indicate relationships between documentWindow and documentText.

we can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer describes a number of behavioral testing methods that can make use of graphs:

- **Transaction flow modeling**. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps
- **Finite state modeling**. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state input).
- **Data flow modeling**. The nodes are data objects, and the links are the transformations that occur to translate one data object into another
- **Timing modeling**. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

2 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present .

An equivalence class represents a set of valid or invalid states for input conditions.

Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.
Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifices a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

3 Boundary Value Analysis

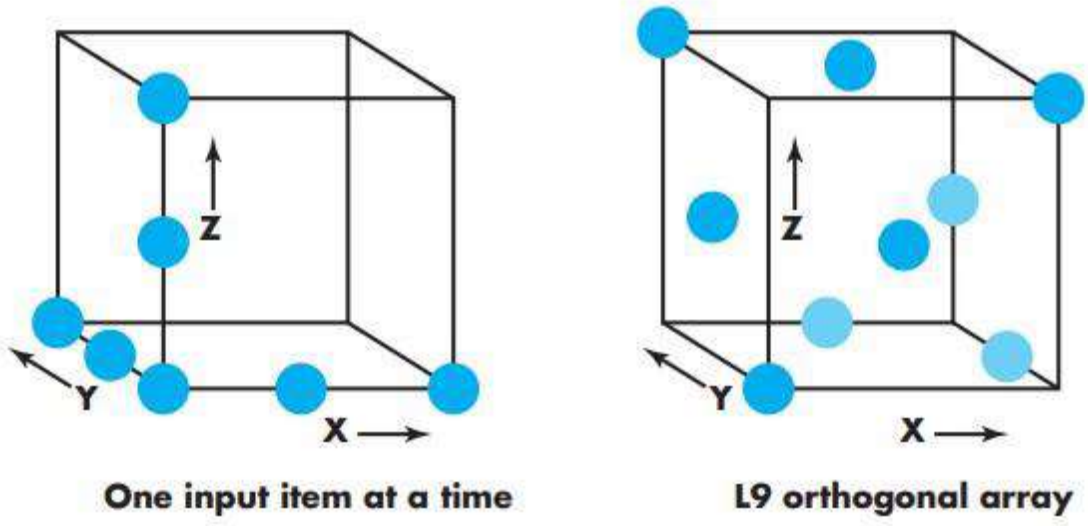
- A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique.
- Boundary value analysis leads to a selection of test cases that exercise bounding values.
- Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

4 Orthogonal Array Testing

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component.
To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, X, Y, and Z.
Each of these input items has three discrete values associated with it. There are 33 = 27 possible test cases. Phadke suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure . Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).



One input item at a time L9 orthogonal array

When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a "balancing property". That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure . Test coverage across the input domain is more complete.
To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:
P1 = 1, send it now
P1 = 2, send it one hour later P1 = 3, send it after midnight
P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.
If a "one input item at a time" testing strategy were chosen, the following sequence of tests

(P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). But these would uncover only single mode faults [Pha97], that is, faults that are triggered by a single parameter.

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is 3 4 5 81, large but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure above assesses the result of tests using the L9 orthogonal array in the following manner:

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

- **Detect and isolate all single mode faults**. A single mode fault is a consistent problem with any level of =any single parameter. For example, if all test cases of factor P1 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 23.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.
- **Detect all double mode faults.** If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault. Indeed, a double mode fault is an indication of pair wise incompatibility or harmful interactions between two test parameters.
- **Multimode faults**. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.

MODEL -BASED TESTING

Model-based testing (MBT) is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases

In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model , as the basis for the design of test cases.

The MBT technique requires five steps:

1. Analyze an existing behavioral model for the software or create one. Recall that a behavioral model indicates how software will respond to external events or stimuli. To create the model, you should perform the steps (1) evaluate all use cases to fully understand the sequence of interaction within the system, (2) identify events that drive the interaction sequence and understand how these events relate to specific objects, (3) create a sequence for each use case, (4) build a UML state diagram for the system and (5) review the behavioral model to verify accuracy and consistency.

2. Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state. The inputs will trigger events that will cause the transition to occur.

3. Review the behavioral model and note the expected outputs as the software makes the transition from state to state. Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created. For each set of inputs (test cases) you specified in step 2, specify the expected outputs as they are characterized in the behavioral model.

4. Execute the test cases. Tests can be executed manually or a test script can be created and executed using a testing tool.

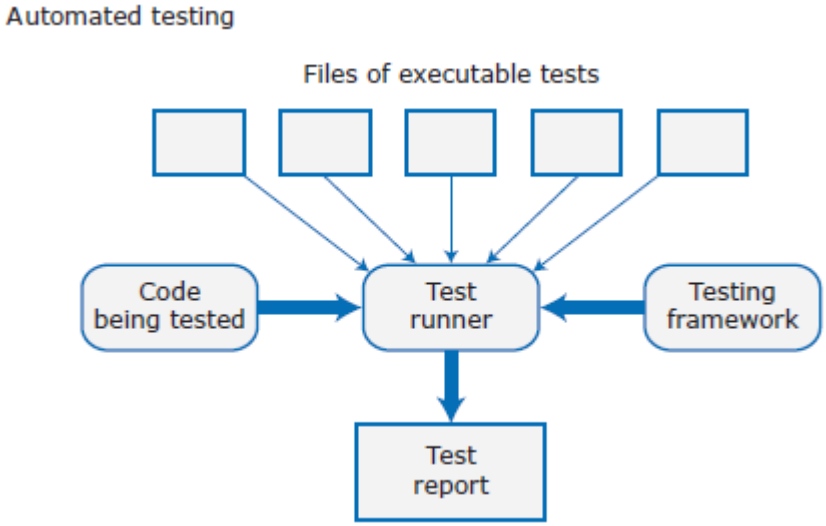
5. Compare actual and expected results and take corrective action as required.
MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

TESTING DOCUMENTATION

Documentation testing can be approached in two phases. The first phase, technical review examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed earlier. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. MBT can be used to ensure that documented behavior and actual behavior coincide. Program usage is then tracked through the documentation.

Test automation



- Automated testing (Figure above) is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result, and a check that the unit returns the expected result.
- we run the test and the test passes if the unit returns the expected result. Normally, we should develop hundreds or thousands of executable tests for a software product.
- The development of automated testing frameworks, such as JUnit for Java in the 1990s, reduced the effort involved in developing executable tests.
- Testing frameworks are now available for all widely used programming languages. A suite of hundreds of unit tests, developed using a framework, can be run on a desktop computer in a few seconds. A test report shows the tests that have passed and failed.
- Testing frameworks provide a base class, called something like "TestCase" that is used by the testing framework. To create an automated test, you define your own test class as a subclass of this TestCase class. Testing frameworks include a way of running all of the tests defined in the classes that are based on TestCase and reporting the results of the tests.

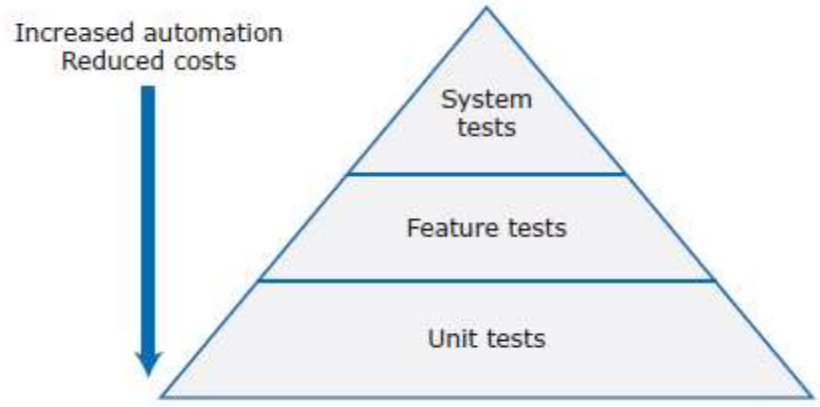
It is good practice to structure automated tests in three parts:
1. *Arrange* You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.
2. *Action* You call the unit that is being tested with the test parameters.
3. *Assert* You make an assertion about what should hold if the unit being tested has executed successfully. If we use equivalence partitions to identify test inputs, we should have several automated tests based on correct and incorrect inputs from each partition.

- the point of automated testing is to avoid the manual checking of test outputs, we can't realistically discover test errors by running the tests. Therefore, we have to **use two approaches to reduce the chances of test errors:**
    - Make tests as simple as possible. The more complex the test, the more likely that it will be buggy. The test condition should be immediately obvious when reading the code.

      Review all tests along with the code that they test. As part of the review process, someone apart from the test programmer should check the tests for correctness.
- Regression testing is the process of re-running previous tests when we make a change to a system.
- This testing checks that the change has not had unexpected side effects. The code change may have inadvertently broken existing code, or it may reveal bugs that were undetected in earlier tests.

- If we use automated tests, regression testing takes very little time. Therefore, after we make any change to your code, even a very minor change, we should always re-run all tests to make sure that everything continues to work as expected.

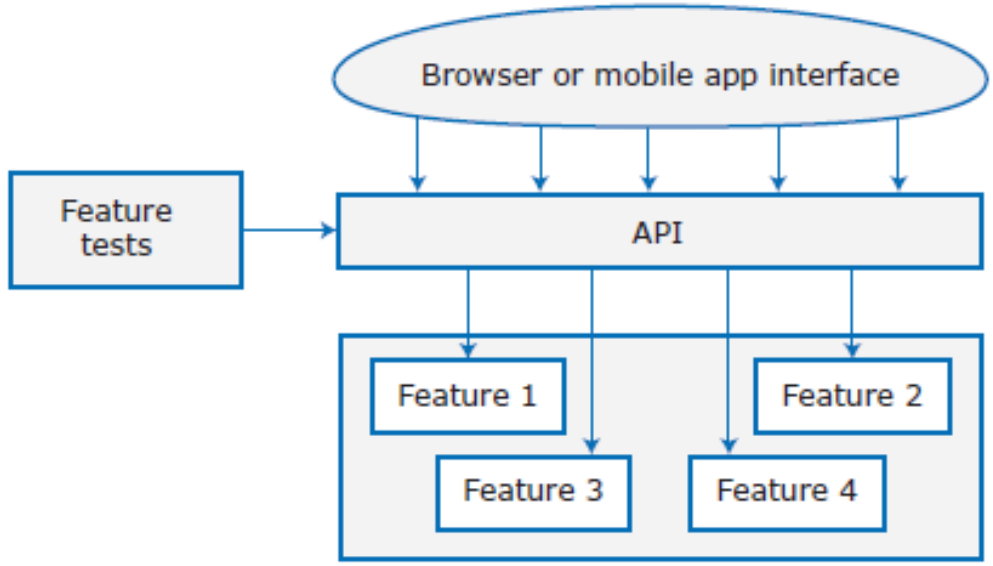
The test pyramid



- Unit tests are the easiest to automate, so the majority of your tests should be unit tests. Mike Cohn, who first proposed the test pyramid , suggests that 70% of automated tests should be unit tests, 20% feature tests (he called these service tests), and 10% system tests (UI tests).
- The implementation of system features usually involves integrating functional units into components and then integrating these components to implement the feature. If you have good unit tests, you can be confident that the individual functional units and components that implement the feature will behave as you expect.
- Generally, users access features through the product's graphical user interface (GUI). However, GUI-based testing is expensive to automate so it is best to use an alternative feature testing strategy.

- This involves designing your product so that its features can be directly accessed through an API, not just from the user interface. The feature tests can then access features directly through the API without the need for direct user interaction through the system's GUI (Figure above).
- Accessing features through an API has the additional benefit that it is possible to re- implement the GUI without changing the functional components of the software.
- For example, a series of API calls may be required to implement a feature that allows a user to share a document with another user by specifying their email address.
- These calls collect the email address and the document identification information, check that the access permissions on the document allow sharing, check that the specified email address is valid and is a registered system user, and add the document to the sharing user's workspace

Feature testing through an API



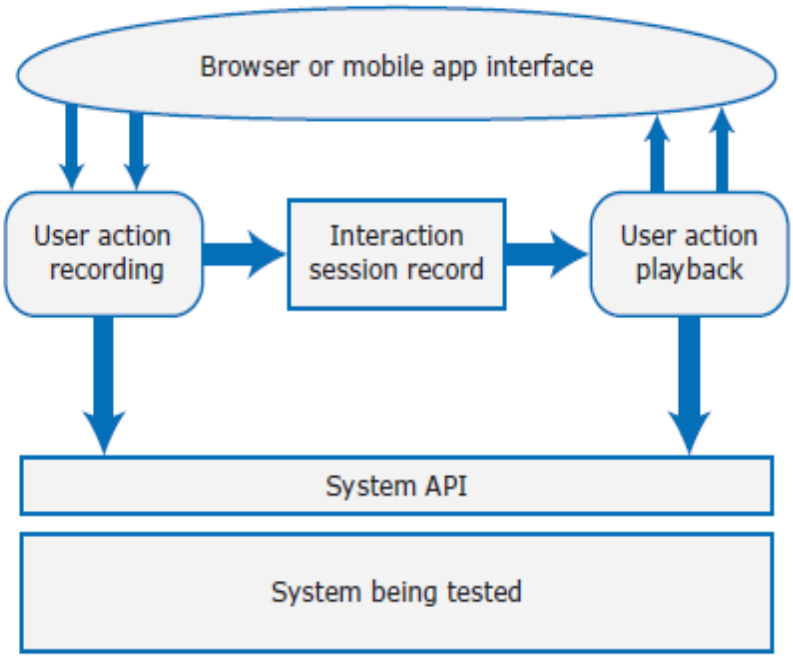
When these calls have been executed, a number of conditions should hold:

- The status of the document is "shared."
- The list of users sharing the document includes the specified email address.
  There have been no deletions from the list of users sharing the document.
- The shared document is visible to all users in the sharing list.

Manual system testing, when testers have to repeat sequences of actions, is boring and prone to errors. In some cases, the timing of actions is important and is practically impossible to repeat consistently. To avoid these problems, testing tools have been developed to record a series of actions and automatically replay them when a

system is retested (Figure 9.7).

Figure 9.7 Interaction recording and playback



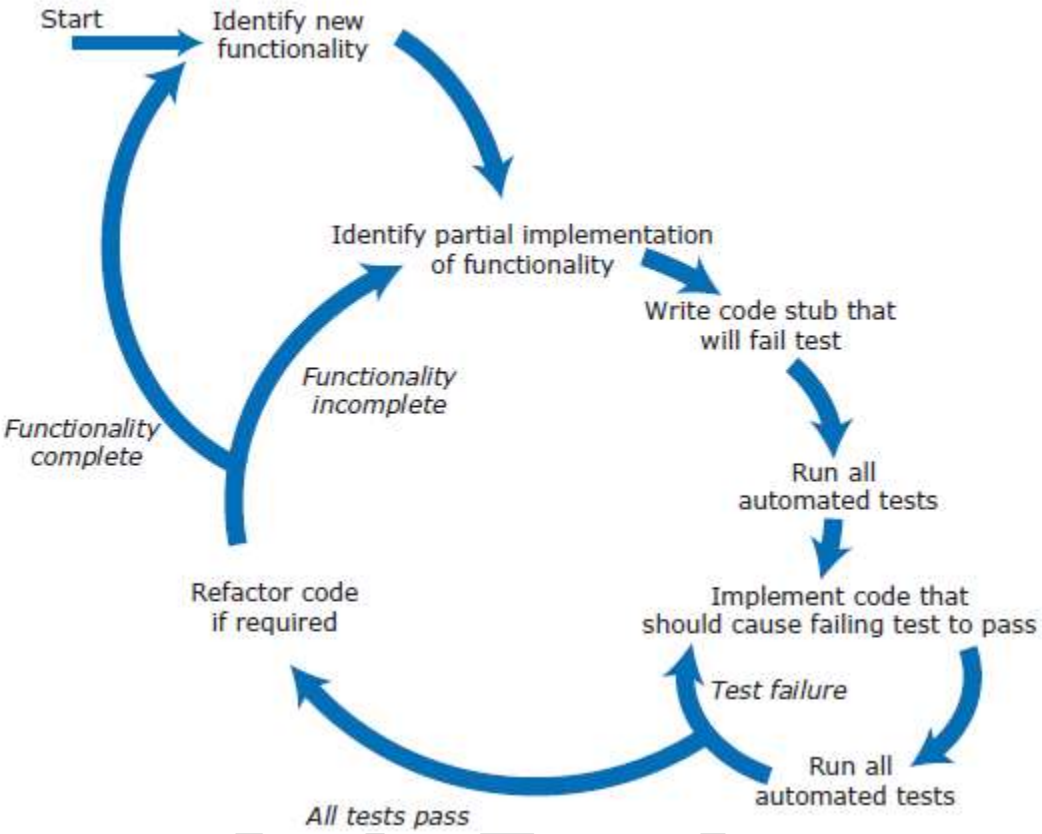
- Interaction recording tools record mouse movements and clicks, menu selections, keyboard inputs, and so on. They save the interaction session and can replay it, sending commands to the application and replicating them in the user's browser interface. These tools also provide scripting support so that you can write and execute scenarios expressed as test scripts. This is particularly useful for cross-browser testing, where you need to check that your software works in the same way with different browsers.
- Automated testing is one of the most important developments in software engineering

Test-driven development

- Test-driven development (TDD) is an approach to program development that is based on the general idea that we should write an executable test or tests for code that are writing before you write the code.

- TDD was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach. Figure 9.8 is a model of the test-driven development process.

Figure 9.8 Test-driven development



- Assume that we have identified some increment of functionality to be implemented.
- Test-driven development relies on automated testing. Every time we add some functionality, we develop a new test and add it to the test suite.
- All of the tests in the test suite must pass before we move on to developing the next increment.
- Test-driven development is an approach in which executable tests are written before the code. Code is then developed to pass the tests.

A disadvantage of test-driven development is that programmers focus on the details of passing tests rather than considering the broader structure of their code and algorithms used.

The benefits of test-driven development are:

1. It is a systematic approach to testing in which tests are clearly linked to section of the program code.
2. The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests..
3. Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.

4. It is argued that TDD leads to simpler code, as programmers only write code that's necessary to pass tests. They don't over engineer their code with complex features that aren't needed.

Table 9.9 Stages of test-driven development

Activity	Description
Identify partial implementation	Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
Write mini-unit tests	Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
Write a code stub that will fail test	Write incomplete code that will be called to implement the mini-unit. You know this will fail.
Run all automated tests	Run all existing automated tests. All previous tests should pass. The test for the incomplete code should fail.
Implement code that should cause the failing test to pass	Write code to implement the mini-unit, which should cause it to operate correctly.
Rerun all automated tests	If any tests fail, your code is incorrect. Keep working on it until all tests pass.
Refactor code if required	If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

DevOps and Code Management

- The ultimate goal of software product development is to release a product to customers. Traditionally, separate teams were responsible for software development, software release, and software support (Figure 1).
- The development team passed a "final" version of the software to a release team. That team then built a release version, tested it, and prepared release documentation before releasing the software to customers.

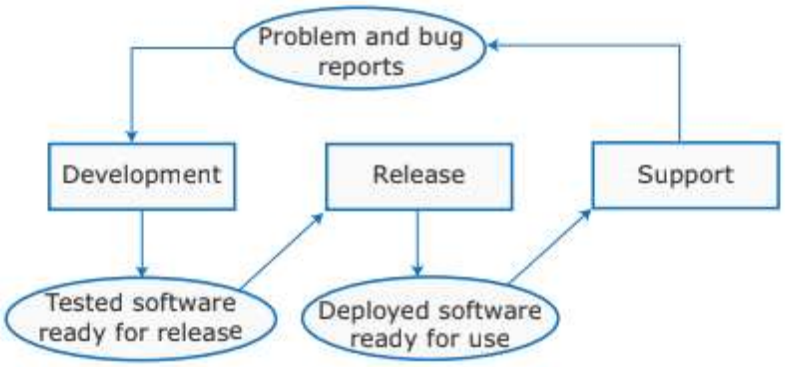


Fig 1 Development, release, and support

- A third team provided customer support. The original development team was sometimes responsible for implementing software changes. Alternatively, the software may have been maintained by a separate maintenance team.
- In these processes, communication delays between the groups were inevitable.
- Development and operations engineers used different tools, had different skill sets, and often didn't understand the other's problems.
- Even when an urgent bug or security vulnerability was identified, it could take several days for a new release to be prepared and pushed to customers.

- Many companies still use this traditional model of development, release, and support.
- However, more and more companies are using an alternative approach called DevOps.
- DevOps (development + operations) integrates development, deployment, and support, with a single team responsible for all of these activities (Figure 2).
- Three factors led to the development and widespread adoption of DevOps:
  1. Agile software engineering reduced the development time for software, but the traditional release process introduced a bottleneck between development and deployment.
  2. Amazon re-engineered their software around services and introduced an approach in which a service was both developed and supported by the same team.
  3. It became possible to release software as a service, running on a public or private cloud. Software products did not have to be released to users on physical media or downloads.

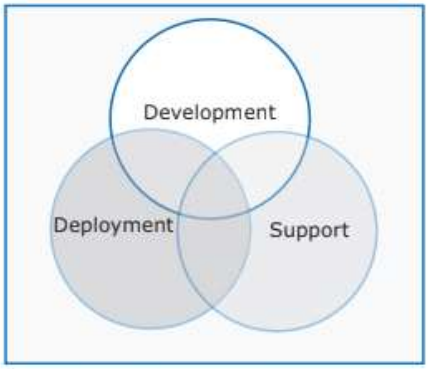


Fig 2 DevOps DevOps Principle

Principle	Explanation
Everyone is responsible for everything.	All team members have joint responsibility for developing, delivering, and supporting the software.
Everything that can be automated should be automated.	All activities involved in testing, deployment, and support should be automated if it is possible to do so. There should be mimimal manual involvement in deploying software.
Measure first, change later.	DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

Table 1 DevOps principles

Benefit	Explanation
Faster deployment	Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.
Reduced risk	The increment of functionality in each release is small so there is less chance of feature interactions and other changes that cause system failures and outages.
Faster repair	DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team was responsible for the problem and to wait for them to fix it.
More productive teams	DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

Table 2 Benefits of DevOps

1: CODE MANAGEMENT

- DevOps depends on the source code management system that is used by the entire team.
- Code management is a set of software-supported practices used to manage an evolving codebase.
- We need code management to ensure that changes made by different developers do not interfere with each other and to create different product versions.

- Code management tools make it easy to create an executable product from its source code files and to run automated tests on that product.
- Source code management, combined with automated system building, is critical for professional software engineering.
- In companies that use DevOps, a modern code management system is a fundamental requirement for "auto- mating everything." Not only does it store the project code that is ultimately deployed, but it also stores all other information that is used in DevOps processes.
  - DevOps automation and measurement tools all interact with the code management system (Figure 3).

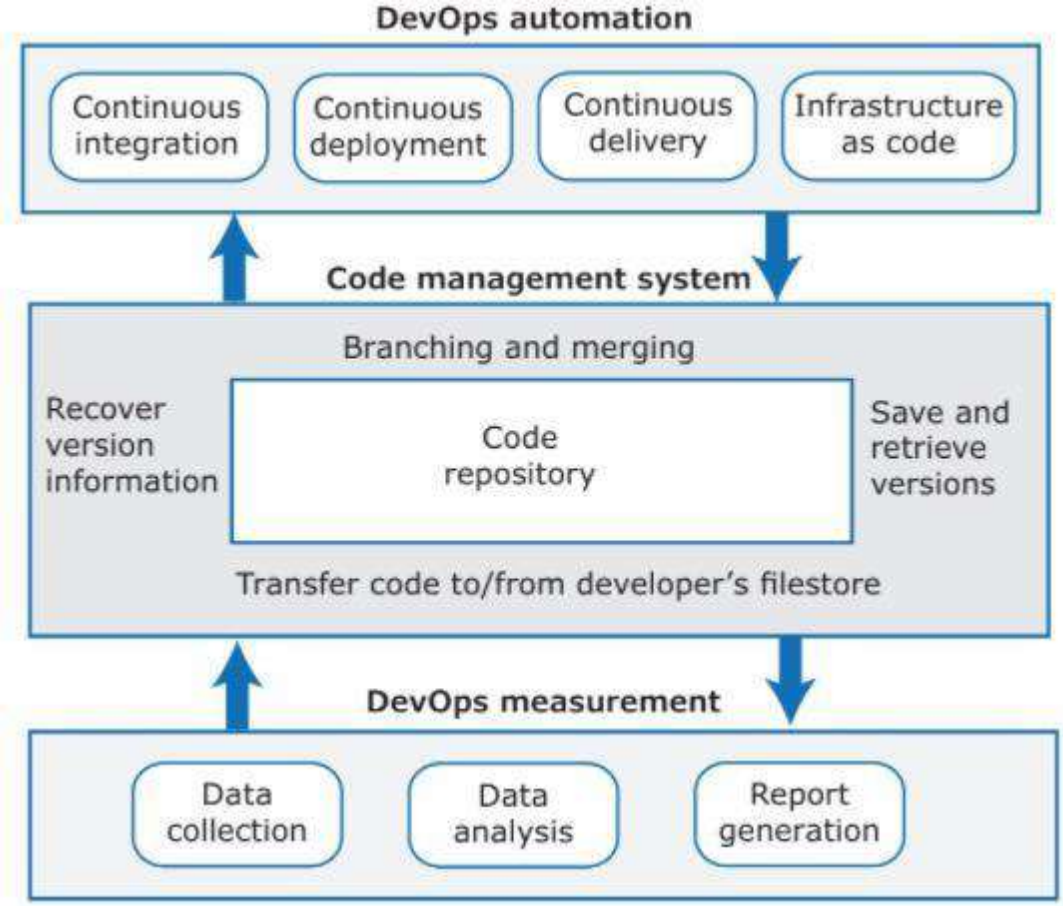


Fig 3: Code management and DevOps

1.1 Fundamentals of source code management

Source code management systems are designed to manage an evolving project codebase to allow different versions of components and entire systems to be stored and retrieved.
Developers can work in parallel without interfering with each other and they can integrate their work with that from other developers.
The code management system provides a set of features that support four general areas:
1. Code transfer Developers take code into their personal file store to work on it; then they return it to the shared code management system.
2. Version storage and retrieval Files may be stored in several different versions, and specific versions of these files can be retrieved.
3. Merging and branching Parallel development branches may be created for concurrent working. Changes made by developers in different branches may be merged.
4. Version information about the different versions maintained in the system may be stored and retrieved. All source code management systems have the general form shown in Figure 3, with a shared repository and a set of

features to manage the files in that repository:

1. All source code files and file versions are stored in the repository, as are other artifacts such as configuration files, build scripts, shared libraries, and versions of tools used. The repository includes a database of information about the stored files, such as version information, information about who has changed the files, what changes were made at what times and so on.
2. The source code management features transfer files to and from the repository and update the information about the different versions of files and their relationships. Specific versions of files and information about these versions can always be retrieved from the repository.

Feature	Description
Version and release identification	Managed versions of a code file are uniquely identified when they are submitted to the system and can be retrieved using their identifier and other file attributes.
Change history recording	The reasons changes to a code file have been made are recorded and maintained.
Independent development	Several developers can work on the same code file at the same time. When this is submitted to the code management system, a new version is created so that files are never overwritten by later changes.
Project support	All of the files associated with a project may be checked out at the same time. There is no need to check out files one at a time.
Storage management	The code management system includes efficient storage mechanisms so that it doesn't keep multiple copies of files that have only small differences.

Table 4 Features of source code management systems

- In 2005, Linus Torvalds, the developer of Linux, revolutionized source code management by developing a distributed version control system (DVCS) called Git to manage the code of the Linux kernel.
- Git was geared to supporting large-scale open-source development.
- It took advantage of the fact that storage costs had fallen to such an extent that most users did not have to be concerned with local storage management.
- Instead of only keeping the copies of the files that users are working on, Git maintains a clone of the repository on every user's computer (Figure 5).
- A fundamental concept in Git is the "master branch," which is the current master version of the software that the team is working on.

we create new versions by creating a new branch, In Figure 5, we can see that two branches have been created in addition to the master branch. When users request a repository clone, they get a copy of the master branch that they can work on independently.

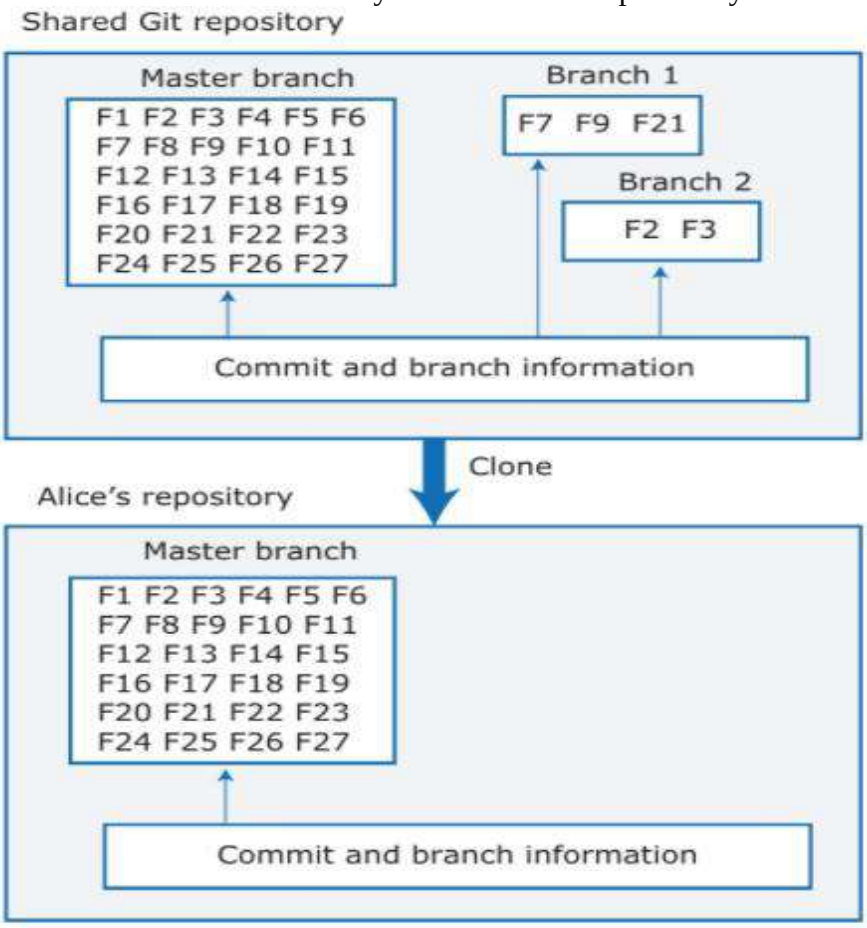


Figure 5 Repository cloning in Git

Git and other distributed code management systems have several advantages over centralized systems:
1. *Resilience* Everyone working on a project has their own copy of the repository. If the shared repository is damaged or subjected to a cyber-attack, work can continue, and the clones can be used to restore the shared repository. People can work offline if they don't have a network connection.
2. Speed committing changes to the repository is a fast, local operation and does not need data to be transferred over the network.
3. Flexibility Local experimentation is much simpler. Developers can safely try different approaches without exposing their experiments to other project members. With a centralized system, this may only be possible by working outside the code management system.

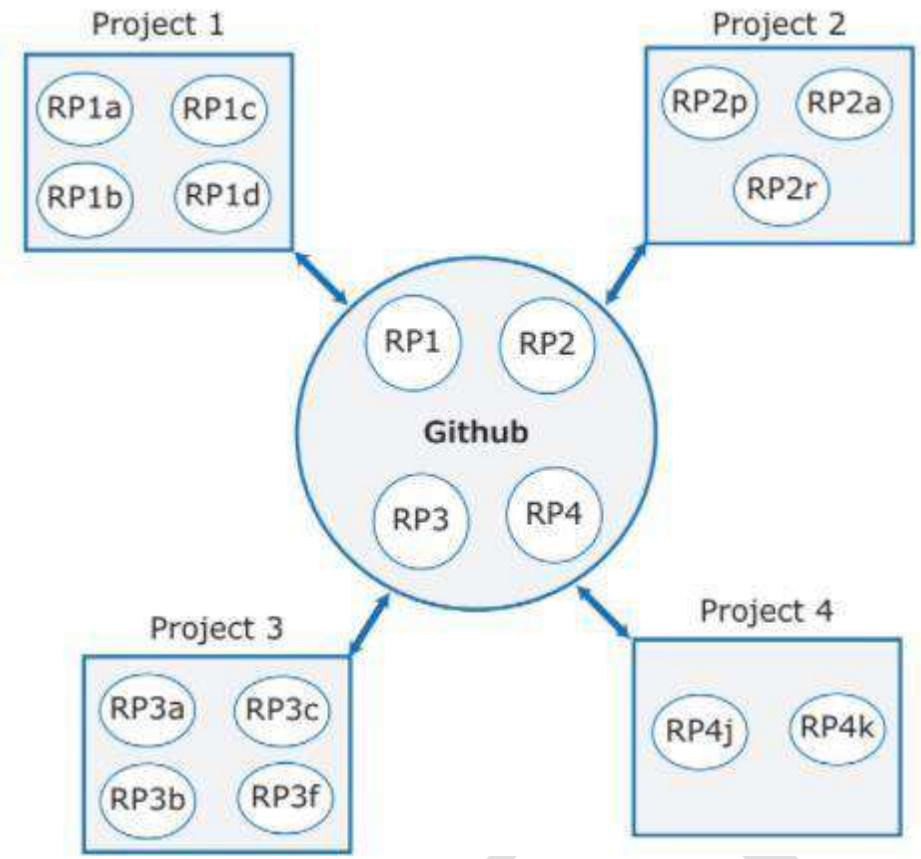


Figure 6 Git repositories

- Most software product companies now use Git for code management.
- For teamwork, Git is organized around the notion of a shared project repository and private clones of that repository held on each developer's computer (Figure 6).
- A company may use its own server to run the project repository. However, many companies and individual developers use an external Git repository provider.
- Several Git repository hosting companies, such as Github and Gitlab, host thousands of repositories on the cloud.

Figure 6 shows four project repositories on Github, RP1–RP4. RP1 is the repository for project 1, RP2 is the repository for project 2, and so on. Each of the developers on each project is identified by a letter (a, b, c, etc.) and has an individual copy of the project repository.
Developers may work on more than one project at a time, so they may have copies of several Git repositories on their computer.
For example, developer a works on Project 1, Project 2, and Project 3, so has clones of RP1, RP2, and RP3.

2: Dev Ops automation

Historically, the processes of integrating a system from independently developed parts, deploying that system in a realistic testing environment, and releasing it were time consuming and expensive. By using DevOps with automated support, however, we can dramatically reduce the time and costs for integration, deployment, and delivery.

- "Everything that can be should be automated" is a fundamental principle of DevOps.
- In addition to reducing the costs and time required for integration, deployment, and delivery, automation makes these processes more reliable and reproducible.

Aspect	Description
Continuous integration	Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.
Continuous delivery	A simulation of the product's operating environment is created and the executable software version is tested.
Continuous deployment	A new release of the system is made available to users every time a change is made to the master branch of the software.
Infrastructure as code	Machine-readable models of the infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform. The software to be installed, such as compilers and libraries and a DBMS, are included in the infastructure model.

Figure 3 showed the four aspects of DevOps automation

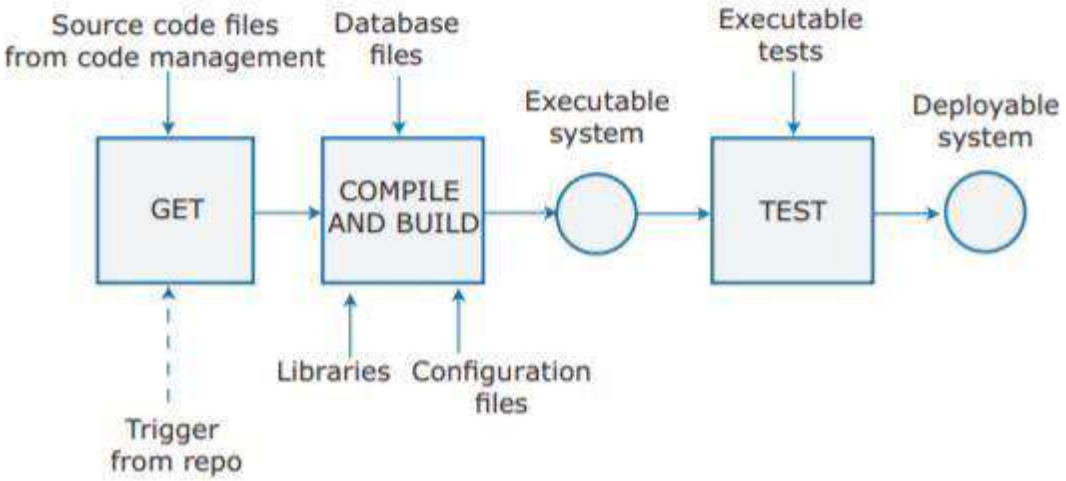
Table Aspects of DevOps automation

2.1 Continuous integration

- System integration (system building) is the process of gathering all of the elements required in a working system, moving them into the right directories, and putting them together to create an operational system. This involves more than compiling the system.
- Continuous integration (CI) means creating an executable version of a software system whenever a change is made to the repository. The CI tool is triggered when a file is pushed to the repo. It builds the system and runs tests on your development computer or project integration server

Figure below Continuous integration

- Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system's shared code repository.



- On completion of the push operation, the repository sends a message to an integration server to build a new version of the product (Figure 9).
- The squares in Figure 9 are the elements of a continuous integration pipeline that is triggered by a repository notification that a change has been made to the master branch of the system.

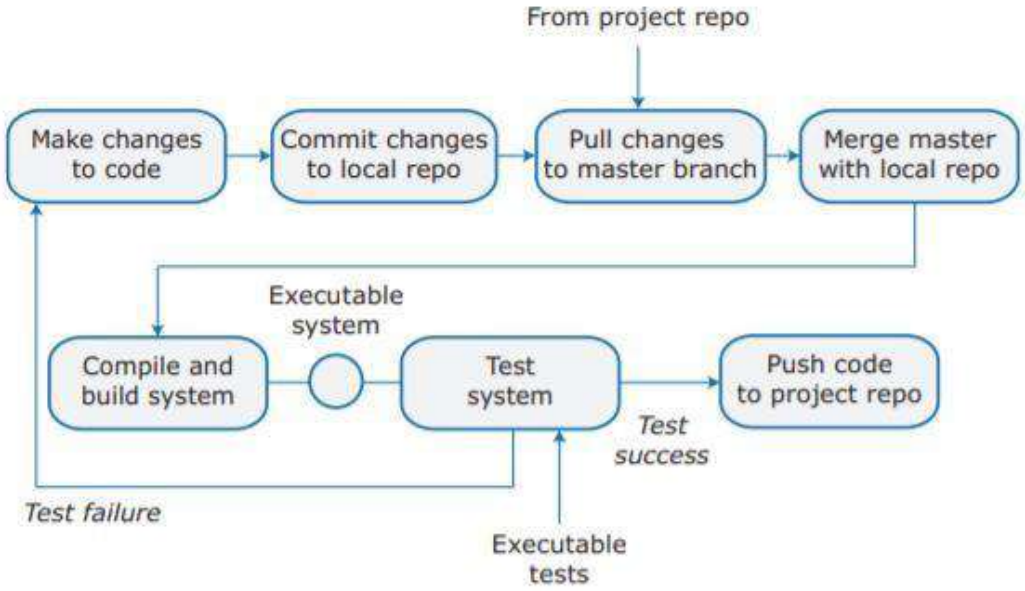


Figure: Local integration

- In a continuous integration environment, developers have to make sure that they don't "break the build." Breaking the build means pushing code to the project repository, which when integrated, causes some of the system tests to fail. This holds up other developers. If this happens, our priority is to discover and fix the problem so that normal development can continue.
- To avoid breaking the build, we should always adopt an "integrate twice" approach to system integration. We should integrate and test on our own computer before pushing code to the project repository to trigger the integration server (Figure10).

- The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system. If we make a small change and some system tests then fail, the problem almost certainly lies in the new code that we have pushed to the project repo. we can focus on this code to find the bug that's causing the problem.
- If we continuously integrate, then a working system is always available to the whole team. This can be used to test ideas and to demonstrate the features of the system to management and customers. Furthermore, continuous integration creates a "quality culture" in a development team. Team members want to avoid the stigma and disruption of breaking the build. They are likely to check their work carefully before pushing it to the project repo.

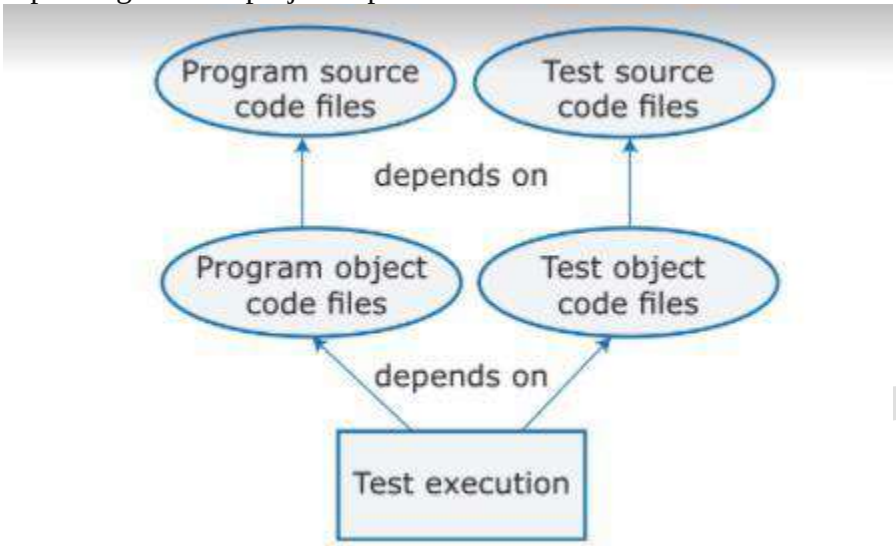


Figure A dependency model

- Continuous integration is effective only if the integration process is fast and developers do not have to wait for the results of their tests of the integrated system.
- However, some activities in the build process, such as populating a database or compiling hundreds of system files, are inherently slow. It is therefore essential to have an automated build process that minimizes the time spent on these activities.
- To understand incremental system building, you need to understand the concept of dependencies.
- Figure above is a dependency model that shows the dependencies for test execution.

- An upward-pointing arrow means "depends on" and shows the information required to complete the task shown in the rectangle at the base of the model. Figure 11 therefore shows that running a set of system tests depends on the existence of executable object code for both the program being tested and the system tests.
- In turn, these depend on the source code for the system and the tests that are compiled to create the object code.
- The first time we integrate a system, the incremental build system compiles all the source code files and executable test files. It creates their object code equivalents, and the executable tests are run. Subsequently, however, object code files are created only for new and modified tests and for source code files that have been modified.

2.2 Continuous delivery and deployment

- Continuous delivery means that, after making changes to a system, we ensure that the changed system is ready for delivery to customers.
- This means that you have to test it in a production environment to make sure that environmental factors do not cause system failures or slow down its performance.
- Continuous delivery does not mean that the software will necessarily be released immediately to users for deployment.

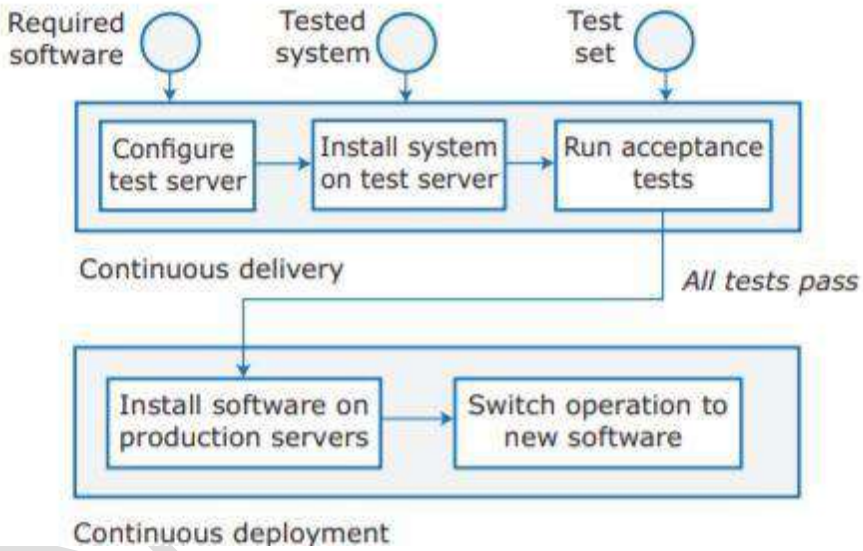


Figure 12 Continuous delivery and deployment

- Figure above illustrates a summarized version of this deployment pipeline, showing the stages involved in continuous delivery and deployment.
- After initial integration testing, a staged test environment is created. This is a replica of the actual production environment in which the system will run.
  - The system acceptance tests, which include functionality, load, and performance tests, are then run to check that the software works as expected.

Benefit	Explanation
Reduced costs	If you use continuous deployment, you have no option but to invest in a completely automated deployment pipeline. Manual deployment is a time-consuming and error-prone process. Setting up an automated system is expensive and takes time, but you can recover these costs quickly if you make regular updates to your product.
Faster problem solving	If a problem occurs, it will probably affect only a small part of the system and the source of that problem will be obvious. If you bundle many changes into a single release, finding and fixing problems are more difficult.
Faster customer feedback	You can deploy new features when they are ready for customer use. You can ask them for feedback on these features and use this feedback to identify improvements that you need to make.
A/B testing	This is an option if you have a large customer base and use several servers for deployment. You can deploy a new version of the software on some servers and leave the older version running on others. You then use the load balancer to divert some customers to the new version while others use the older version. You can measure and assess how new features are used to see if they do what you expect.

Table 6

Benefits of continuous deployment

2.3 Infrastructure as code

- Rather than manually updating the software on a company's servers, the process can be automated using a model of the infrastructure written in a machine-process able language.
- Configuration management (CM) tools, such as Puppet and Chef, can automatically install software and services on servers according to the infrastructure definition. The CM tool accesses a master copy of the software to be installed and pushes this to the servers being provisioned (Figure 13).
- When changes have to be made, the infrastructure model is updated and the CM tool makes the change to all servers.
- Defining our software infrastructure as code is obviously relevant to products that are delivered as services.

- The product provider has to manage the infrastructure of their services on the cloud. However, it is also relevant if software is delivered through downloads.

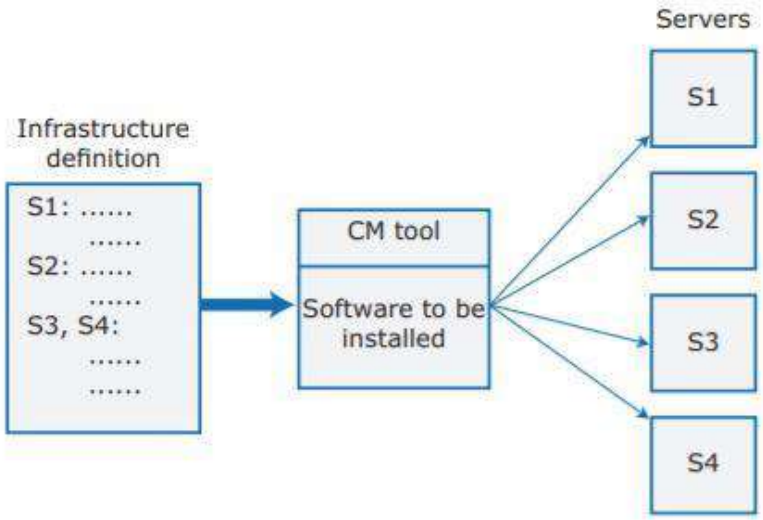


Figure : Infrastructure as code

Defining our infrastructure as code and using a configuration management system solve two key problems of continuous deployment:

1. Our testing environment must be exactly the same as your deployment environment. If you change the deployment environment, you have to mirror those changes in your testing environment.
2. When we change a service, we have to be able to roll that change out to all of our servers quickly and reliably. If there is a bug in our changed code that affects the system's reliability, we have to be able to seamlessly roll back to the older system.

The business benefits of defining our infrastructure as code are lower costs of system management and lower risks of unexpected problems arising when infrastructure changes are implemented. These benefits stem from four fundamental characteristics of infrastructure as code, shown in Table 7

Characteristic	Explanation
Visibility	Your infrastructure is defined as a stand-alone model that can be read, discussed, understood, and reviewed by the whole DevOps team.
Reproducibility	Using a configuration management tool means that the installation tasks will always be run in the same sequence so that the same environment is always created. You are not reliant on people remembering the order that they need to do things.
Reliability	In managing a complex infrastructure, system administrators often make simple mistakes, especially when the same changes have to be made to several servers. Automating the process avoids these mistakes.
Recovery	Like any other code, your infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems, you can easily revert to an older version and reinstall the environment that you know works.

Table 10.7 Characteristics of infrastructure as code

3 DevOps measurement

After you have adopted DevOps, you should try to continuously improve your DevOps process to achieve faster deployment of better-quality software. This means you need to have a measurement program in place in which you collect and analyze product and process data. By making measurements over time, you can judge whether or not you have an effective and improving process
Measurements about software development and use fall into four categories:
1. **Process measurements** :collect and analyze data about your development, testing, and deployment processes.
2. **Service measurements** ;collect and analyze data about the software's performance, reliability, and acceptability to customers.
3. **Usage measurements** : collect and analyze data about how customers use your product.
4. **Business success measurements** : collect and analyze data about how your product contributes to the overall success of the business.

MODULE 4 NOTES

SOFTWARE PROJECT MANAGEMENT

Software project management is an essential part of software engineering.

- The success criteria **for project management** obviously vary from projectto project, but, for most projects, **important goals are:**
    1. to deliver the software to the customer at the agreed time;
    2. to keep overall costs within budget
    3. to deliver software that meets the customer's expectations;
    4. to maintain a coherent and well-functioning development team.
- Software engineering is different from other types of engineering in anumber of ways:
    1. The product is intangible.
    2. Large software projects are often "one-off" projects.
    3. Software processes are variable and organization-specific.
- It is impossible to write a standard job description for a softwareproject manager.
- *Some of the most **important factors that affect how softwareprojects are managed are:***
    1. Company size
    2. Software customers
    3. Software size
    4. Software type
    5. Organizational culture
    6. Software development processes
- The **fundamental project management activities** that are common to all organizations:
    1. **Project planning** Project managers are responsible for planning, estimating, andscheduling project development and assigning people to tasks.
    2. **Risk management** Project managers have to assess the risks that may affect aproject, monitor these risks, and take action when problems arise.
    3. **People management** Project managers are responsible for managing a team ofpeople. They have to choose people for their team and establish ways of workingthat lead to effective team performance.
    4. **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.
    5. **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 theobjectives of the project and how it will be carried out. It usually includes cost andschedule estimates and justifies why the project contract should be awarded to a particular organization or a team

RISK MANAGEMENT

- Risk management is one of the most important jobs for a projectmanager.
- Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, andthen taking action to avoid these risks.
- Risks can be categorized according to type of risk (technical, organizational, etc.)

- ***Classification of risks according to what these risks affect:***
  1. Project risks affect the project schedule or resources. An example of a project risk is the loss of an experienced system architect.
  2. **Product risks** affect the quality or performance of the software beingdeveloped. An example of a product risk is the failure of a purchased component to perform as expected.
  3. **Business risks** affect the organization developing or procuring thesoftware. For example, a competitor introducing a new product is a business risk.

- For large projects, you should record the results of the risk analysis ina risk register along with a consequence analysis. This sets out the consequences of the risk for the project, product, and business.

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 company 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.
Software tool underperformance	Product	Software tools that 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.

Fig: Examples of common project, product, and business risks

- Effective risk management makes it easier to cope with problems andto ensure that these do not lead to unacceptable budget or scheduleslippage.
- For small projects, formal risk recording may not be required, but theproject manager should be aware of them.
- The specific risks that may affect a project depend on the project andthe organizational environment in which the software is being developed.
- Software risk management is important because of the inherentuncertainties

  in software development.

- An outline of the process of risk management is presented inFigure. It involves several **stages:**
  1. **Risk identification** □ You should identify possible project, product,and business risks.
  2. **Risk analysis** □ You should assess the likelihood and consequences ofthese risks.
  3. **Risk planning** □ You should make plans to address the risk, either by avoiding it or by minimizing its effects on the project.
  4. **Risk monitoring** □ You should regularly assess the risk and your plans for risk mitigation and revise these plans when you learn moreabout the risk.

The risk management process is an iterative process that continues throughout a project.

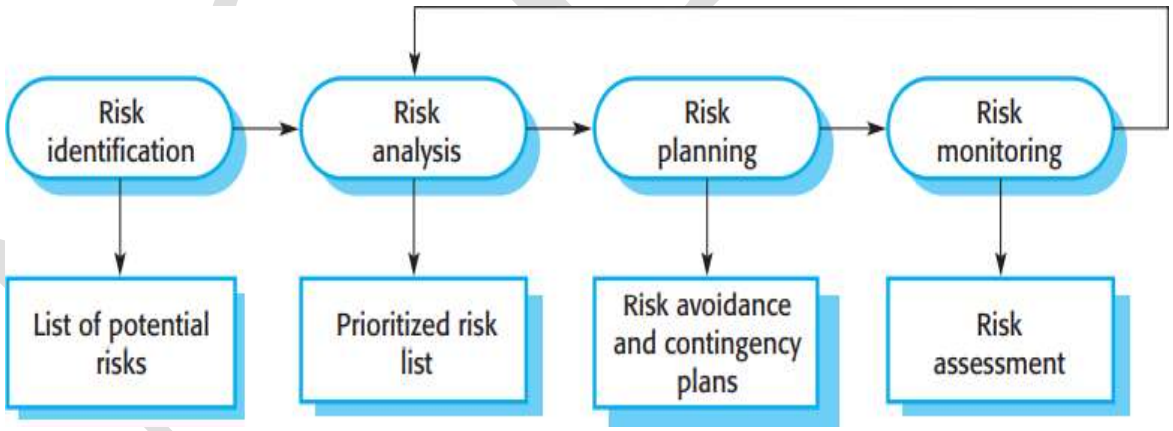


Fig: The Risk Management Process

Risk Identification:

- Risk identification is the first stage of the risk management process.
- It is concerned with identifying the risks that could pose a majorthreat to the software engineering process, the software being developed, or the development organization.
- Risk identification may be a team process in which a team getstogether to brainstorm possible risks.
- As a starting point for risk identification, a checklist of different typesof risk may be used.
- **6 types of risk** may be included in a risk checklist:

1. **Estimation risks** □ arise from the management estimates of the resources required to build the system.
2. **Organizational risks** □ arise from the organizational environment wherethe software is being developed.
3. **People risks** □ are associated with the people in the development team.
4. **Requirements risks** □ come from changes to the customer requirements and the process of managing the requirements change.
5. **Technology risks** □ come from the software or hardware technologies that are used to develop the system.
6. **Tools risks** □ come from the software tools and other support softwareused to develop the system.

Risk type	Possible risks
Estimation	1. The time required to develop the software is underestimated. 2. The rate of defect repair is underestimated. 3. The size of the software is underestimated.
Organizational	4. The organization is restructured so that different management are responsible for the project. 5. Organizational financial problems force reductions in the project budget.
People	6. It is impossible to recruit staff with the skills required. 7. Key staff are ill and unavailable at critical times. 8. Required training for staff is not available.
Requirements	9. Changes to requirements that require major design rework are proposed. 10. Customers fail to understand the impact of requirements changes.
Technology	11. The database used in the system cannot process as many transactions per second as expected. 12. Faults in reusable software components have to be repaired before these components are reused.
Tools	13. The code generated by software code generation tools is inefficient. 14. Software tools cannot work together in an integrated way.

Figure 22.3 Examples of different types of risk

Risk Analysis:

- During the risk analysis process, you have to consider each identified risk and make a judgment about the probability and seriousness of that risk.
- It is not possible to make precise, numeric assessment of the probability and seriousness of each risk.
- You should assign the risk to one of a number of bands:
    1. The probability of the risk might be assessed as insignificant, low, moderate, high, or very high.
    2. The effects of the risk might be assessed as catastrophic (threaten the survival of the project),serious (would cause major delays), tolerable (delays are within allowed contingency), or insignificant.
- You may then tabulate the results of this analysis process using a table ordered according

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (5).	Low	Catastrophic
It is impossible to recruit staff with the skills required (6).	High	Catastrophic
Key staff are ill at critical times in the project (7).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused (12).	Moderate	Serious
Changes to requirements that require major design rework are proposed (9).	Moderate	Serious
The organization is restructured so that different managements are responsible for the project (4).	High	Serious
The database used in the system cannot process as many transactions per second as expected (11).	Moderate	Serious
The time required to develop the software is underestimated (1).	High	Serious
Software tools cannot be integrated (14).	High	Tolerable
Customers fail to understand the impact of requirements changes (10).	Moderate	Tolerable
Required training for staff is not available (8).	Moderate	Tolerable
The rate of defect repair is underestimated (2).	Moderate	Tolerable
The size of the software is underestimated (3).	High	Tolerable
Code generated by code generation tools is inefficient (13).	Moderate	Insignificant

Figure 22.4 Risk types and examples

- Both the probability and the assessment of the effects of a risk may change as more information about the risk becomes available and asrisk management plans are implemented. You should therefore update this table during each iteration of the risk management process.
- Once the risks have been analyzed and ranked, you should assesswhich of these risks are most significant.
- In general, catastrophic risks should always be considered, as shouldall serious risks that have more than a moderate probability of occurrence.

RiskPlanning:

- The risk planning process develops strategies to manage the key risksthat threaten the project.
- For each risk, you have to think of actions that you might take to minimize the disruption to the project if the problem identified in therisk occurs.
- You should also think about the information that you need to collectwhile monitoring the project so that emerging problems can be detected before they become serious.
- In risk planning, you have to ask "what-if" questions that consider both individual risks, combinations of risks, and external factors thataffect these risks. For example, questions that you might ask are:

1What if several engineers are ill at the same time?

1. What if an economic downturn leads to budget cuts of 20% for the project?
2. What if the performance of open-source software is inadequate and the

   only expert on that open-source software leaves?
3. What if the company that supplies and maintains software componentsgoes out of business?
4. What if the customer fails to deliver the revised requirements as predicted?

Based on the answers to these "what-if" questions, you may devise strategies for managing the risks

- The **possible risk management strategies** fall into 3 categories:
  1. **Avoidance strategies** □ Following these strategies means that the probability that the risk will arise is reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components.
  2. **Minimization strategies** □ Following these strategies means that the impact of the risk is reduced. An example of a risk minimization strategy isthe strategy for staff illness.
  3. **Contingency plans** □ Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems.

- The strategies used in critical systems ensure reliability, security, andsafety, where you must avoid, tolerate, or recover from failures.

- It is best to use a strategy that avoids the risk.
- If this is not possible, you should use a strategy that reduces thechances that the risk will have serious effects.
- Finally, you should have strategies in place to cope with the risk if it arises. These should reduce the overall impact of a risk on the projector product.

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 automated code generation.

Figure 22.5 Strategies to help manage risk

RiskMonitoring

- Risk monitoring is the process of checking that your assumptions about the product, process, and business risks have not changed.
- You should regularly assess each of the identified risks to decidewhether or not that risk is becoming more or less probable.
- You should also think about whether or not the effects of the risk have changed.

- To do this, you have to look at other factors, such as the number of requirements change requests, which give you clues about the risk probability and its effects. These factors are dependent on the typesof risk.

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships among team members; high staff turnover.
Requirements	Many requirements change requests; customer complaints.
Technology	Late delivery of hardware or support software; many reported technology problems.
Tools	Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, and so on.

Figure 22.6 Risk indicators

- You should monitor risks regularly at all stages in a project.
- At every management review, you should consider and discuss eachof the key risks separately.
- You should decide if the risk is more or less likely to arise and if theseriousness and consequences of the risk have changed.

MANAGING PEOPLE

- The people working in a software organization are its greatest assets.
- It is expensive to recruit and retain good people.
- Software managers have to ensure that the engineers working on a project are as productive as possible.
- It is important that software project managers understand the technical issues that influence the work of software development.
- Software engineers often have strong technical skills but may lack the softer skills that enable them to motivate and lead a project development team.
- As a project manager, you should be aware of the potential problems of people management and should try to develop people management skills.

- **4 critical factors that influence the relationship between a manager and the people** that he or she manages:

  1. **Consistency** □ All the people in a project team should be treated in a comparable way. No one expects all rewards to be identical, but people shouldnot feel that their contribution to the organization is undervalued.
  2. **Respect** □ Different people have different skills, and managers should respectthese differences.
  3. **Inclusion** □ People contribute effectively when they feel that others listen to them and take account of their proposals. It is important to develop a workingenvironment where all views, even those of the least experienced staff, are considered.
  4. **Honesty** □ As a manager, you should always be honest about what is goingwell and what is going badly in the team. You should also be honest about your level of technical knowledge and be willing to defer to staff with moreknowledge when necessary.

MotivatingPeople:

- As a project manager, you need to motivate the people who workwith you so that they will contribute to the best of their abilities.
- In practice, **motivation** means organizing work and its environment toencourage people to work as effectively as possible.
- To provide this encouragement, you should understand a little aboutwhat motivates people.
- People are motivated by satisfying their needs. These needs arearranged in a series of levels, as shown in Figure.

Figure 22.7 Human needs hierarchy

1. **To satisfy social needs**, you need to give people time to meet their co-workers and provide places for them to meet. This is relatively easy whenall of the members of a development team work in the same place.
   Social networking systems and teleconferencing can be used for remotecommunications.

2. **To satisfy esteem needs**, you need to show people that they are valued by the organization. Public recognition of achievements is a simple and effective way of doing this.

3. Finally, **to satisfy self-realization needs**, you need to give people responsibility for their work, assign them demanding (but not impossible) tasks, and provide opportunities for training and development where people can enhance their skills. Training is an important motivating influence as people like to gain new knowledgeand learn new skills.

The lower levels of this hierarchy represent fundamental needs for food,sleep, and so on, and the need to feel secure in an environment.

- **Social need** is concerned with the need to feel part of a social grouping.
- **Esteem need** represents the need to feel respected by others, and **self-realization need** is concerned with personal development.
- People need to satisfy lower-level needs such as hunger before the moreabstract, higher-level needs.
- People working in software development organizations are not usually hungry, thirsty, or physically threatened by their environment. Therefore,making sure that peoples' social, esteem, and self-realization needs are satisfied is most important from a management point of view.
- Maslow's model of motivation takes an exclusively personal viewpoint

on motivation.

- It does not take adequate account of the fact that people feel themselves to be part of an organization, a professional group, andone or more cultures.
- Being a member of a cohesive group is highly motivating for mostpeople.
- Therefore, as a manager, you also have to think about how a group asa whole can be motivated.

Case study: Motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of six developers that can develop new products based on the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team, and creative new ideas are developed. The team decides to develop a system that a user can initiate and control the alarm system from a cell phone or tablet computer. However, some months into the project, Alice notices that Dorothy, a hardware expert, starts coming into work late, that the quality of her work is deteriorating, and, increasingly, that she does not appear to be communicating with other members of the team.

Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

After some initial denials of any problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity to use these skills. Basically, she is working as a C programmer on the alarm system software.

While she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

Figure 22.8 Individual motivation

- **<u>Psychological personality type also influences motivation.</u>**
- Bass and Dunteman (Bass and Dunteman 1963) identified **3classifications for professional workers:**

1. **Task-oriented people** □ who are motivated by the work they do. In software engineering, these are people who are motivated by the intellectual challenge of software development.
2. **Self-oriented people** □ who are principally motivated by personal success and recognition. They are interested in software developmentas a means of achieving their own goals. They often have longer-termgoals and they wish to be successful in their work to help realize these goals.
3. **Interaction-oriented people** □ who are motivated by the presenceand actions of co-workers.

Research has shown that interaction-oriented personalities usuallylike to work as part of a group, whereas task-oriented and self- oriented people usually prefer to act as individuals.

• **People Capability Maturity Model (P-CMM)**

 is a framework for assessing how well organizations manage the development of their staff. It highlights best practice in people management and provides abasis for organizations to improve their people management processes. It is best suited to large rather than small, informal companies.

- Effective communication is achieved when communications are two-way andthe people involved can discuss issues and information and establish a common understanding of proposals and problems.
- All this can be done through meetings, although these meetings are oftendominated by powerful personalities.
- Informal discussions when a manager meets with the team for coffee aresometimes more effective.
- Wikis and blogs allow project members and external stakeholders to exchange information, irrespective of their location. They help manageinformation and keep track of discussion threads, which often becomeconfusing when conducted by email.
- You can also use instant messaging and teleconferences, which can be easilyarranged, to resolve issues that need discussion.

TEAMWORK

- As it is impossible for everyone in a large group to work together on asingle problem, large teams are usually split into a number of smallergroups.
- Each group is responsible for developing part of the overall system.
- The best size for a software engineering group is 4 to 6 members, andthey should never have more than 12 members.
- When groups are small, communication problems are reduced.

- Putting together a group that has the right balance of technical skills,experience, and personalities is a critical management task.
- A good group is cohesive and thinks of itself as a strong, single unit.
- The people involved are motivated by the success of the group as wellas by their own personal goals.
- In a **cohesive group**, members think of the group as more importantthan the individuals who are group members.
  - They are loyal to the group.
  - They identify with group goals and other group members.
  - They attempt to protect the group, as an entity, from outside interference. This makes the group robust and able to cope with problems and unexpected situations.

- An effective way of making people feel valued and part of a group isto make sure that they know what is going on.

Case study: Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As her company is developing a new product, she takes the opportunity to involve all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing, and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an "away day" for the group where the team spends two days on "technology updating." Each team member prepares an update on a relevant technology and presents it to the group. This is an offsite meeting, and plenty of time is scheduled for discussion and social interaction.

Figure 22.9 Group cohesion

- The **benefits of creating a cohesive group** are:
    1. The group can establish its own quality standards.
    2. Individuals learn from and support each other.
    3. Knowledge is shared.
    4. Refactoring and continual improvement is encouraged.
- Good project managers should always try to encourage groupcohesiveness.
- They may try to establish a sense of group identity by naming the groupand establishing a group identity and territory.
- One of the most effective ways of promoting cohesion is **to be inclusive** i.e., you should treat group members as responsible and trustworthy, andmake information freely available.
- Given a stable organizational and project environment, the **3 factors that have the biggest effect on team working are:**
    1. The people in the group (**Selecting group members**)
    2. The way the group is organized (**Group organizations**)
    3. Technical and managerial communications (**Group communications**)

Selecting Group Members:

- A manager or team leader's job is to create a cohesive group and organize that group so that they work together effectively.
- This task involves selecting a group with the right balance of technical skillsand personalities.
- Technical knowledge and ability should not be the only factor used toselect group members.
- People who are motivated by the work are likely to be the strongesttechnically.
- People who are self-oriented will probably be best at pushing the workforward to finish the job.
- People who are interaction-oriented help facilitate communications withinthe group.
- The project manager has to control the group so that individual goalsdo not take precedence over organizational and group objectives.

- This control is easier to achieve if all group members participate in each stage of the project.
- Individual initiative is most likely to develop when group members aregiven instructions without being aware of the part that their task plays in the overall project.
- If all the members of the group are involved in the design from the start, they are more likely to understand why design decisions have been made. They may then identify with these decisions rather thanoppose them

Case study: Group composition

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly being promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

Alice—self-oriented
Brian—task-oriented
Chun—interaction-oriented
Dorothy—self-oriented
Ed—interaction-oriented
Fiona—task-oriented
Fred—task-oriented
Hassan—interaction-oriented

Figure 22.10 Group composition

Group Organization

- The way a group is organized affects the group's decisions, the ways information is exchanged, and the interactions between the development group and external project stakeholders.
- Project managers are often responsible for selecting the people in the organization who will join their software engineering team.
- Getting the best possible people in this process is very important as poor selection decisions may be a serious risk to the project.
- Key factors that should influence the selection of staff are education and training, application domain and technology experience, communication ability, adaptability, and problem solving ability.

Important organizational questions for project managers

include thefollowing:

1. Should the project manager be the technical leader of the group?
2. Who will be involved in making critical technical decisions, and how will these decisions be made? Will decisions be made by the system architect orthe project manager or by reaching consensus among a wider range of team members?
3. How will interactions with external stakeholders and senior companymanagement be handled?
4. How can groups integrate people who are not co-located?
5. How can knowledge be shared across the group?

Informal Groups	Hierarchical Groups
1. Small programming groups are usually organized. 2. Group leader gets involved in the software development with the other group members. 3. The group as a whole discusses the work to be carried out, and tasks are allocated according to ability and experience. 4. More senior group members may be responsiblefor the architectural design. 5. Detailed design and implementation is the responsibility of the team member who is allocatedto a particular task. 6. Groups are very successful, particularly when most group members are experienced and competent. Such a group makes decisions which improves cohesiveness and performance. 7. With no experienced engineers to direct the work, the result can be a lack of coordination between group members and, possibly, eventual project failure.	1. Group leader is at the top of the hierarchy. 2. Group leader has more formal authority than the group members and so can direct their work. 3. There is a clear organizational structure. 4. Decisions are made toward the top of the hierarchy and implemented by people lower down. 5. Communications are primarily instructions from senior staff; the people at lower levels of the hierarchy have relatively little communication with the managers at the upper levels. 6. These groups can work well when a well-understood problem can be easily broken down into software components that can be developed indifferent parts of the hierarchy. 7. This grouping allows for rapid decision making.

- In software development, effective team communications at all levels isessential:
    1. Changes to the software often require changes to several parts of the system, and this requires discussion and negotiation at all levels in the hierarchy.
    2. Software technologies change so fast that more junior staff may know more about new technologies than experienced staff. Top-down communications may mean that the project manager does not find out about the opportunities of using these new technologies. More junior staff may become frustrated because of what they see as old-fashioned technologies being used for development.
- A major challenge facing project managers is the difference in technicalability between group members.
- i.e., adopting a group model that is based on individual experts can posesignificant risks.

GroupCommunications:

- It is absolutely essential that group members communicate effectivelyand efficiently with each other and with other project stakeholders.
- Good communication also helps strengthen group cohesiveness.
- Group members:
    1. Exchange information on the status of their work, the design decisions thathave been made, and changes to previous design decisions.
    2. Resolve problems that arise with other stakeholders and inform thesestakeholders of changes to the system, the group, and delivery plans.

3. Come to understand the motivations, strengths, and weaknesses of otherpeople in the group.

- The effectiveness and efficiency of communications are influenced by:

1. **Group size** □ As a group gets bigger, it gets harder for members to communicate effectively. The number of one-way communication links isn * (n − 1), where n is the group size.
2. **Group structure** □ People in informally structured groups communicate more effectively than people in groups with a formal, hierarchical structure.
3. **Group composition** □ People with the same personality may clash, and, asa result, communications can be inhibited.
4. **The physical work environment** □ The organization of the workplace is amajor factor in facilitating or inhibiting communications.
5. **The available communication channels** □ There are many different formsof communication—face to face, email messages, formal documents, telephone, and technologies such as social networking and wikis.

Configuration management

- Configuration management (CM) is concerned with the policies, processes, and tools for managing changingsoftware systems.
- You need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- Versions implement proposals for change, corrections of faults, and adaptations for different hardware andoperating systems.

- Several versions may be under development and in use at the same time.
- If you don't have effective configuration management procedures in place, you may waste effort modifyingthe wrong version of a system, delivering the wrong version of a system to customers, or forgetting where the software source code for a particular version of the system or component is stored.
- Configuration management is useful for individual projects as it is easy for one person to forget whatchanges have been made.
- It is essential for team projects where several developers are working at the same time on a softwaresystem.
- The configuration management system provides team members with access to the system being developedand manages the changes that they make to the code

The configuration management of a software system product involves fourclosely related activities (Figure 1):

1. **Version control**: This involves keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
2. **System building**: This is the process of assembling program components, data,and libraries, then compiling and linking these to create an executable system.
3. **Change management**: This involves keeping track of requests for changes to delivered software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes shouldbe implemented.
4. **Release management**: This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

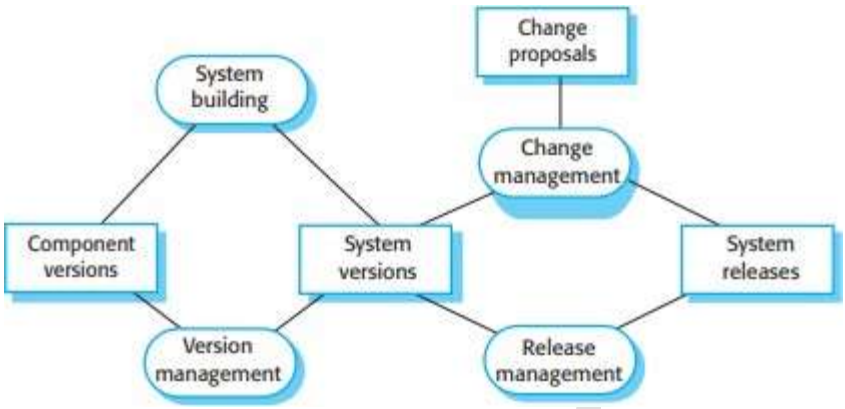


Figure : Configuration management activities

CONFIGURATION MANAGEMENT (SCM)

- Change is inevitable when computer software is built and can lead to confusion when you and other members of a software team are working on a project.
- Configuration management is the art of identifying,organizing, and controlling modifications to the software being built by aprogrammingteam.
- The goal is to maximize productivity by minimizing mistakes.
- SCM activities are developed to (1) identify change, (2) control change,(3) ensure that change is being properly implemented, and (4) report changes toothers who may have an interest.
- There are four fundamental sources of change:
  - (i) New business or market conditions dictate changes in product requirements or business rules.
  - (ii) New stakeholder needs demand modification of data produced by informationsystems, functionality delivered by products, or services deliveredby a computer-based system.
  - (iii) Reorganization or business growth/downsizing causes changes in projectpriorities or software engineering team structure.
  - (iv) Budgetary or scheduling constraints cause a redefinition of the system orproduct.

Elements of SCM

Four important elements that should exist when a configuration management system is developed:

(a) Component elements —A set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.

(b) Process elements —A collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.

(c) Construction elements —A set of tools that automate the construction ofsoftware by ensuring that the proper set of validated components (i.e., thecorrect version) have been assembled.

(d) Human elements —A set of tools and process features (encompassing otherCM elements) used by the software team to implement effective SCM.

Baseline

- A baseline is a software configuration management concept that helps you to control change without seriously impeding justifiable change.
- A specification or product that has been formally reviewed and agreed upon, thatthereafter serves as the basis for further development, and that can be changed onlythrough formal change control procedures.
- Before a software configuration item becomes a baseline, change may be madequickly and informally.
  However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

Version and release management

- Invent identification scheme for system versions
- Plan when new system version is to be produced
- Ensure that version management procedures and tools are properly applied
- Plan and distribute new system releases

Versions/variants/releases

- *Version* An instance of a system which is functionally distinct in some way from other system instances
- *Variant* An instance of a system which is not functionally identical but non-functionally distinct from other instances of a system
- *Release* An instance of a system which is distributed to users outside of the development team

Version identification

- Procedures for version identification should define an unambiguous way of identifying component versions
- Three basic techniques for component identification
    - Version numbering
    - Attribute-based identification
    - Change-oriented identification

Version numbering

- Simple naming scheme uses a linear derivation
  e.g. V1, V1.1, V1.2, V2.1, V2.2 etc.
- However, actual derivation structure may be a tree or a network rather than a sequence
- Names are not meaningful.
- Hierarchical naming scheme may be better

Version derivation structure

Attribute-based identification

- Attributes can be associated with a version with the combination of attributes identifying that
  version
- Examples of attributes are Date, Creator, Programming Language, Customer, Status etc.
- More flexible than an explicit naming scheme for version retrieval;
- Supports queries when looking for versions
- Can cause problems with uniqueness

- Awkward - needs an associated name for easy reference
- Examples of attributes are
    1. Customer
    2. Development language
    3. Development status
    4. Hardware platform
    5. Creation date

Change-oriented identification

- Integrates versions and the changes made to create these versions
- Used for systems rather than components
- Each proposed change has a change set that describes changes made to implement that change
- Change sets are applied in sequence so that, in principle, a version of the system that incorporates an arbitrary set of changes may be created

Release management

- Releases must incorporate changes forced on the system by errors discovered by users and
  by hardware changes
- They must also incorporate new system functionality
- Release planning is concerned with when to issue a system version as a release

System releases

- Not just a set of executable programs
- May also include
    - Configuration files defining how the release is configured for a particular installation
    - Data files needed for system operation
    - An installation program or shell script to install the system on target hardware
    - Electronic and paper documentation
    - Packaging and associated publicity

Release decision making

- Must plan when to distribute a system release
- Preparing and distributing a release is expensive
- Factors:
    - technical quality
    - competition, marketing requirements
    - customer change requests

System release strategy

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. However, minor system faults may be repaired by issuing patches (often distributed over the Internet) that can be applied to the current release of the system.
Lehman's fifth law (see Chapter 27)	This suggests that the increment of functionality which is included in each release is approximately constant. Therefore, if there has been a system release with significant new functionality, then it may have to be followed by a repair release.
Competition	A new system release may be necessary because a competing product is available.
Marketing requirements	The marketing department of an organisation may have made a commitment for releases to be available at a particular date.
Customer change proposals	For customised systems, customers may have made and paid for a specific set of system change proposals and they expect a system release as soon as these have been implemented.

Release creation

- Collect all files and documentation
- Configuration descriptions / instructions/ scripts
- Documented to allow re-creation

Release Documentation

- Whenever a system release is actually produces , it must be documented to ensure that it can be re created exactly in future
- This is mainly used in customized long term embedded systems
- To document a release we have to record the specific versions of source code components which were used to create executable code
- There need to record versions of OS, libraries, compilers and other tools used to build software

Empirical Cost Estimation Model: COCOMO Model

- The Constructive Cost Model (COCOMO) is a procedural software costestimation model developed by Barry W Boehm.
- COCOMO is used to estimate size, effort and duration based on the cost of the software
- COCOMO consists of a hierarchy of three increasingly detailed and accurate forms
- The first level, Basic COCOMO is good for quick, early, rough order of magnitude estimates of software costs.
- But its accuracy is limited due to its lack of factors to account for difference in project attributes (Cost Drivers).
- Intermediate COCOMO takes these Cost Drivers into account.
- Detailed COCOMO additionally accounts for the influence of individual project phases.

Basic COCOMO

- Basic COCOMO computes software development effort (and cost) as a function of program size.
- Program size is expressed in estimated thousands of source lines of code (SLOC, KLOC).
- COCOMO applies to three classes of software projects:
- Organic projects - "small" teams with "good" experience working with "less than rigid" requirements
- Semi-detached projects - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
- Embedded projects - developed within a set of "tight" constraints. It is also combination of organic and semi-detached projects.(hardware, software, operational, ...)

- The basic COCOMO equations take the form

Effort Applied (E) = $a_b(KLOC)^{b_b}$

Development Time (D) = $c_b(\text{Effort Applied})^{d_b}$

People required (P) = Effort Applied / Development Time

where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for project.

- The coefficients a_b, b_b, c_b and d_b are given in the following table:

Software project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

- Basic COCOMO is good for quick estimate of software costs
- However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, etc

Intermediate COCOMO

- Intermediate COCOMO computes software development effort as function of
- program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes

This extension considers a set of four "cost drivers", each with a number of subsidiary attributes:-

(a) Product attributes

Required software reliability extent

Size of application database

Complexity of the product

(b) Hardware attributes

Run-time performance constraints

Memory constraints

Volatility of the virtual machine environment Required turnabout time

(c) Personnel attributes

Analyst capability

Software engineering capability Applications experience

Virtual machine experience Programming language experience

(d) Project attributes

Use of software tools

Application of software engineering methods

Required development schedule

Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value).

- The product of all effort multipliers results is an effort adjustment factor (EAF).

Typical values for EAF range from 0.9 to 1.4

- The Intermediate Cocomo formula now takes the form:

$$E=a_i(KLoC)(b_i)(EAF)$$

where E is the effort applied in person months,

KLoC is the estimated number of thousands of delivered lines of code for the project,

EAF is the factor calculated above.

- The coefficient a_i and the exponent b_i are given in the above table.
- The Development time D calculation uses E in the same way as in the Basic COCOMO.

Detailed COCOMO

- Detailed COCOMO incorporates all characteristics of the intermediate version

Software project	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

The detailed model uses different effort multipliers for each cost driver attribute.

- These Phase Sensitive effort multipliers are each to determine the amount of effort required to complete each phase.
- In detailed COCOMO, the whole software is divided into different modules and then we apply COCOMO in different modules to estimate effort and then
- sum the effort.

- The effort is calculated as a function of program size and a set of cost drivers are given according to each phase of the software life cycle.

The Six phases of detailed COCOMO are:-

A)planning and requirements

B)system design

C)detailed design

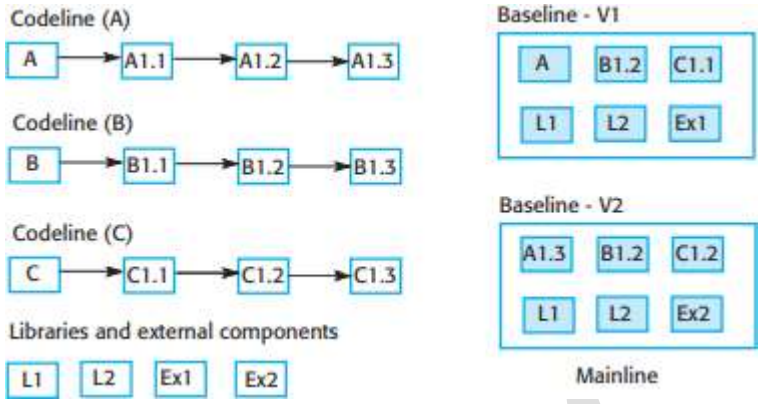
D)module code and test

E)integration and test

F) Cost Constructive model

Version Management

- Version management is the **process of keeping track of different versions of software components and the systems in which these components are used.**
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- In other words, version management is the process of managing codelines and baselines.
- A **codeline** is a sequence of versions of source code, with later versions in the sequence derived from earlier versions.
- Codelines normally apply to components of systems so that there are different versions of each component.
- A **baseline** is a definition of a specific system.
- The baseline specifies the component versions that are included in the system and identifies the libraries used, configuration files, and other system information.
- In the figure different baselines use different versions of the components from each codeline.
- In the diagram, boxes representing components are shaded in the baseline definition to indicate that these are actually references to components in a codeline. The mainline is a sequence of system versions developed from an original baseline



- Baselines may be specified using a configuration language in which you define what components should be included in a specific versionof a system.
- It is possible to explicitly specify an individual component version(X.1.2, say) or simply to specify the component identifier (X).
- If you simply include the component identifier in the configurationdescription, the most recent version of the component should be used.
- Baselines are important because you often have to re-create an individual version of a system
- **Version control (VC) systems** identify, store, and control access to the different versions of components.
- There are **two types of modern version control system**:

1. **Centralized systems**, where a single master repository maintains allversions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.

2. **Distributed systems**, where multiple versions of the component repository exist at the same time. Git, is a widely used example of adistributed VC system.

Centralized and distributed VC systems provide comparable functionality but implement this functionality in different ways. **Key features** of these systems include:

1. Version and release identification: Managed versions of a component are assigned unique identifiers when they are submittedto the system. These identifiers allow different versions of the same component to be managed, without changing the component name. Versions may also be assigned attributes, with the set of attributes used to uniquely identify each version.

2. Change history recording: The VC system keeps records of the changes that have been made to create a new version of a component from an earlier version.

3. Independent development: Different developers may be working on the same component at the same time. The version control system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.

4. Project support: A version control system may support the development of several projects, which share components. It is usually possible to check in and check out all of the files associated with a project rather than having to work with one file ordirectory at a time.

5. Storage management: Rather than maintain separate copies of all versions of a component, the version control system may useefficient mechanisms to ensure that duplicate copies of identical files are not maintained. Where there are only small differences between files, the VC system may store these differences rather than maintain multiple copies of files. A specific version may be automatically re-created by applying the differences to a master version

Most software development is a team activity, so several team members often work on the samecomponent at the same time.

- It's important to avoid situations where changes interfere with each other.
- The project repository maintains the "master" version of all components, which is used to create baselines for system building. When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
- When they have completed their changes, the changed components are returned (checked-in) to therepository.
- However, **centralized and distributed VC systems support independent development of shared components in different ways.**
- In **centralized** systems (FIGURE 3**)**, developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- When their changes are complete, they check-in the components back to the repository. This creates a new component version that may then be shared.
- If two or more people are working on a component at the same time, each must check out the component from the repository.
- If a component has been checked out, the version control system warns other users wanting to check outthat component that it has been checked out by someone else.
- The system will also ensure that when the modified components are checked in, the different versions are assigned different version identifiers and are stored separately.

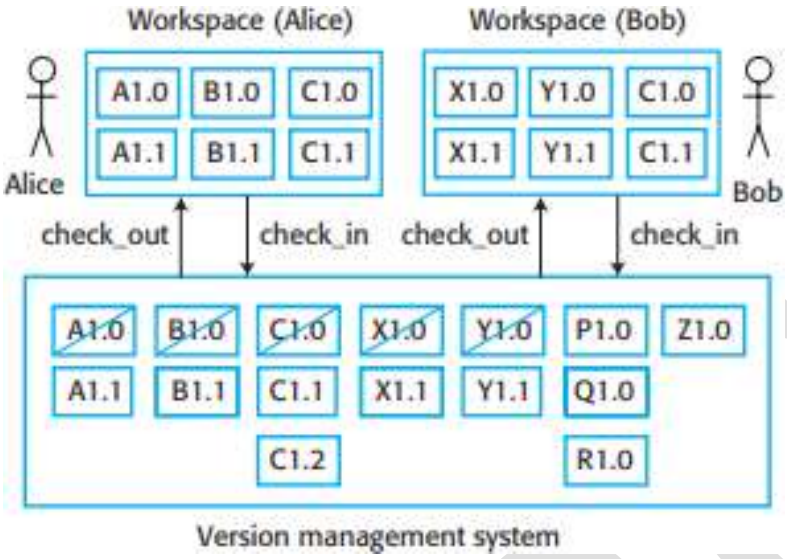


Figure: Check in and Check out from a centralized version repository.

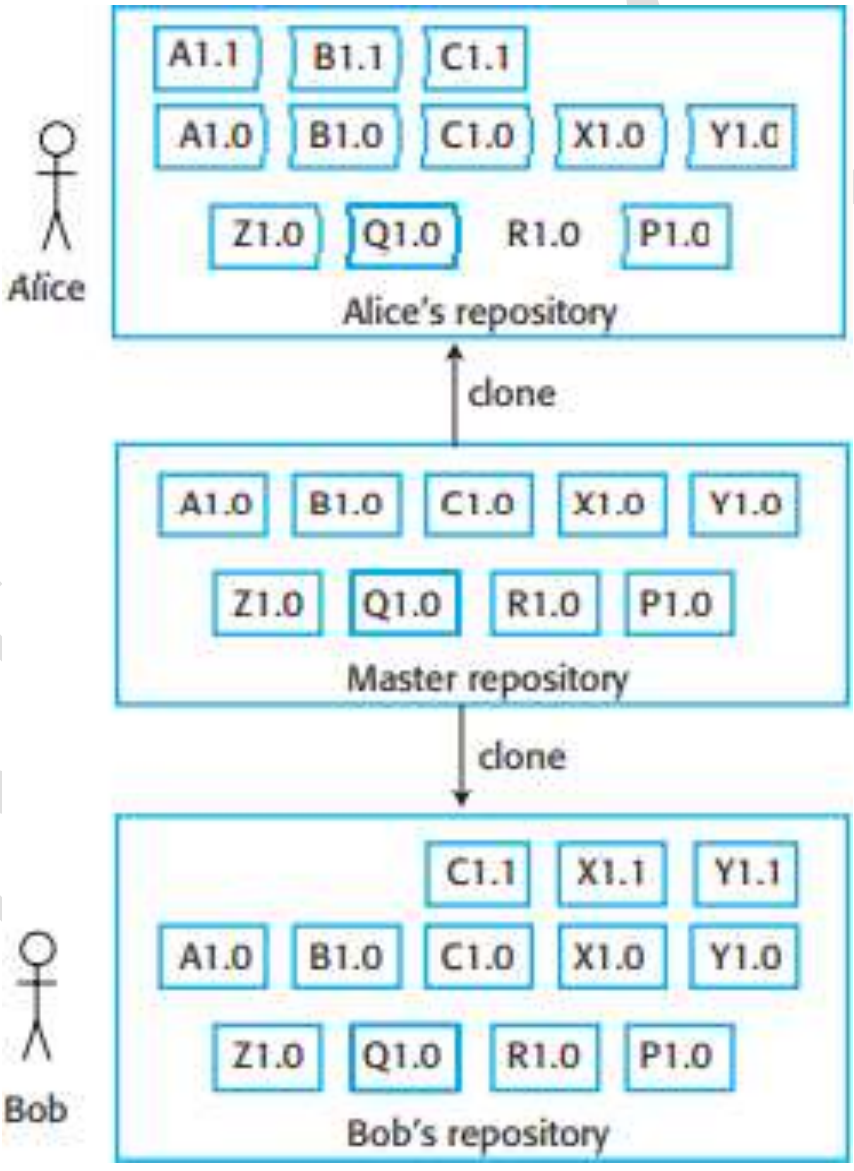
- In a **distributed VC system**, such as Git, a different approach is used.
- A "master" repository is created on a server that maintains the code produced by the development team.
- Instead of simply checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on his or her computer.
- Developers work on the files required and maintain the new versions on their privaterepository on their own computer.
- When they have finished making changes, they "commit" these changes and updatetheir private server repository.
- They may then "push" these changes to the project repository or tell the integrationmanager that changed versions are available.
- He or she may then "pull" these files to the project repository (see Figure 4). In this example, both Bob and Alice have cloned the project repository and have updated files.
- They have not yet pushed these back to the project repository.

This model of development has a number ofadvantages:

1. It provides a backup mechanism for the repository. If the repository is corrupted, work can continue and the project repository can be restored from local copies.
2. It allows for offline working so that developers can commit changes if they donot have a network connection.
3. Project support is the default way of working. Developers can compile and test the entire system on their local machinesand test the changes they have made.

Distributed version control is essential for open- source development where several people may be working simultaneously on the same system without any central coordination.

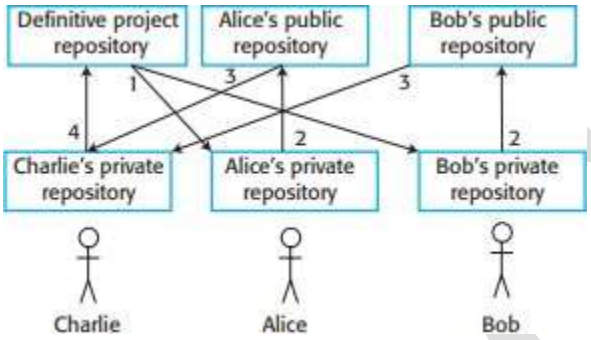
There is no way for the open-source system "manager" to know when changes will be made.



- In this case, as well as a private repository on their own computer, developers also maintain apublic server repository to which they push newversions of components that they have changed.
- It is then up to the open-source system "manager" to decide when to pull these changesinto

the definitive system.

- This organization is shown in figure 5.



- A consequence of the independent development of the same component is that **codelines may branch**.
- Rather than a linear sequence of versions that reflect changes to the component over time, there may be

several independent sequences, as shown in Figure 6.

- This is normal in system development, where different developers work independently on different versions

of the source code and change it in different ways.

- It is generally recommended when working on a system that a new branch should be created so that changes do not accidentally break a working system.
- At some stage, it may be necessary to **merge codeline branches** to create a new version of a component thatincludes all changes that have been made.
- This is also shown in Figure 6, where component versions 2.1.2 and 2.3 are merged to create version 2.4.
- If the changes made involve completely different parts of the code, the component versions may be mergedautomatically by the version control system by combining the code changes.
- This is the normal mode of operation when new features have been added.
- These code changes are merged into the master copy of the system. However, the changes made bydifferent developers sometimes overlap.

- The changes may be incompatible and interfere with each other. In this case, a

developer has to check forclashes and make changes to the components to resolve the incompatibilities between the different versions.

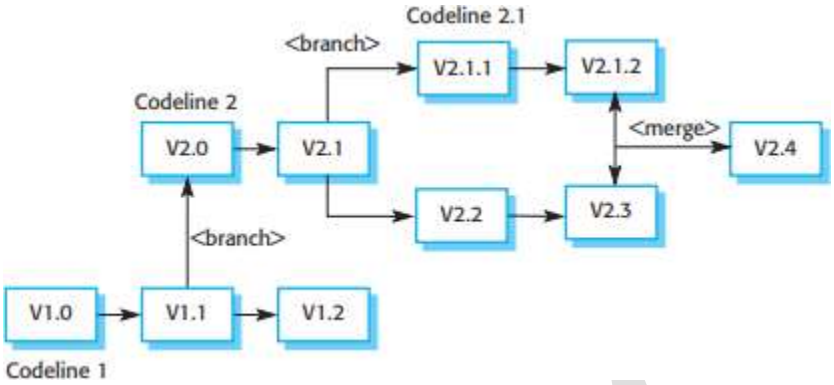


Figure 6: Branching and Merging

- When version control systems were first developed, **storage management** was one of their most important functions. Disk space was expensive, and it was important to minimize the disk space used by the different copies of components.
- Instead of keeping a complete copy of each version, the **system stores a list of differences (deltas) between one version andanother.**
- By applying these to a master version (usually the most recent version), a target version can be re-created. This is illustrated in

Figure 7.

- When a new version is created, the system simply stores a delta, a list of differences, between the new version and the olderversion used to create that new version.
- In Figure 7, the shaded boxes represent earlier versions of a component that are automatically re-created from the most recentcomponent version.
- Deltas are usually stored as lists of changed lines, and, by applying these automatically, one version of a component can be created

from another.

- As the most recent version of a component will most likely be the one used, most systems store that version in full. The deltasthen define how to re-create earlier system versions.
- One of the problems with **a delta-based** approach to storage management is that it can take a long time to apply all of the deltas.
- As disk storage is now relatively cheap, Git uses an alternative, faster approach. Git does not use deltas but applies a standard

compression algorithm to stored files and their associated meta-information. It does not store duplicate copies of files.

- Retrieving a file simply involves decompressing it, with no need to apply a chain of

operations.

- Git also uses the notion of packfiles where several smaller files are combined into an indexed single file. This reduces the overheadassociated with lots of small files. Deltas are used within packfiles to further reduce their size

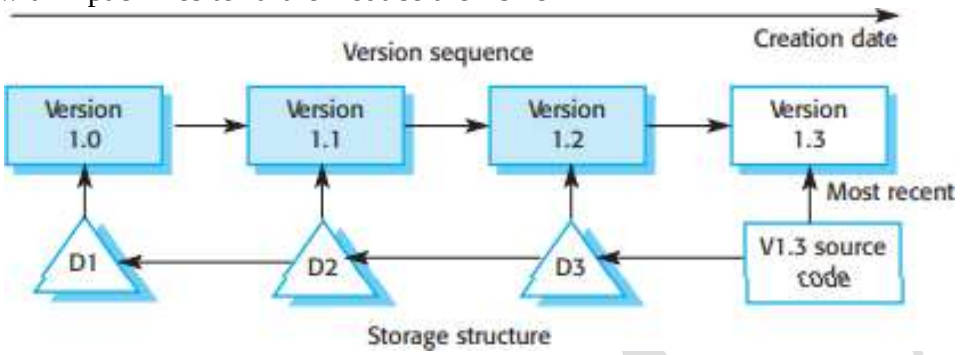
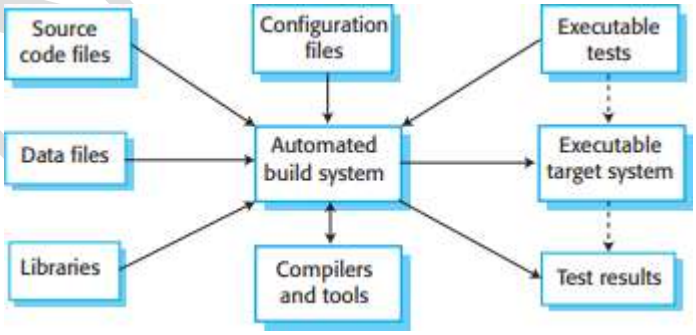


Figure: Storage management using deltas

System building

- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, and other information.
- System-building tools and version control tools must be integrated as the build process takes component versions from the repository managed by the version control system.
- System building involves assembling a large amount of information about the software and its operating environment.
- Therefore, it always makes sense to use an automatedbuild tool to create a system build (Figure 8).
- source code files that are involved in the build are not enough. You may have to link these with externally provided libraries, data files (such as a file of error messages), and configuration files that define the target installation.
- You may have to specify the versions of the compiler and other software tools that are to be used in the build. Ideally, you should be able to build a complete system with



- Figure 8: System building

Tools for system integration and building include some or all of the following features:

1. Build script generation: The build system should analyze the program that is being built, identify dependent components, and automatically generate a build script (configuration file). The system shouldalso support the manual creation and editing of build scripts.

2. Version control system integration: The build system should check out the required versions of components from the version control system.

3. Minimal recompilation: The build system should work out what source code needs to be recompiled andset up compilations if required.

4. Executable system creation: The build system should link the compiled object code files with each other and with other required files, such as libraries and configuration files, to create an executable system.

5. Test automation: Some build systems can automatically run automated tests using test automation tools such as JUnit. These check that the build has not been "broken" by changes.

6. Reporting: The build system should provide reports about the success or failure of the build and the testsstat have been run.

7. Documentation generation: The build system may be able to generate release notes about the build andsystem help pages.

- The build script is a definition of the system to be built.
- It includes information about components and their dependencies, and the versions of tools used to compile and link the system.
- The configuration language used to define the build script includes constructs to describe the system components to be included in thebuild and their dependencies.

• Building is a complex process, which is potentially error-prone, as three different systemplatforms may be involved (Figure 9):

1. **The development system**, which includes development tools such as compilers and source code editors. Developers check out code from the version control system into a private workspace before making changes to the system. They may wish to build a version of a system for testing in their development environment before committing changes that they have madeto the version control system.

2. The **build server**, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system. All of the system developers check in codeto the version control system on the build server for system building.

3. The **target environment**, which is the platform on which the system executes. This may be the same type of computer that is used for the development and build systems. However, for real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g., a cell phone). For large systems, the target environment may include databases and other application systems that cannot be installed on development machines. In these situations, it is not possible to build and test the system on the development computer or on the build server

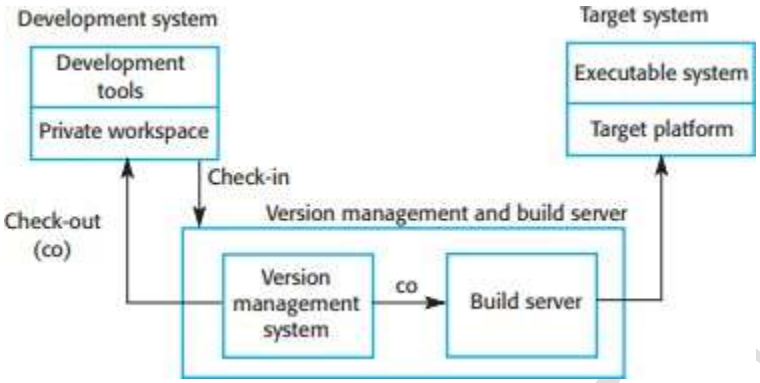


Figure: Development, build and target platforms

- Agile methods recommend that very frequent system builds should be carried out, with automated testing used to discover software problems. Frequent builds are part of a process of continuous integration as shown in Figure 10.
- In keeping with the agile methods notion of making many small changes, **continuous integration** involves rebuilding the mainline frequently, after small source code changes have been made.
- The **steps in continuous integration** are:

1. Extract the mainline system from the VC system into the developer's private workspace.
2. Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken, and you should inform whoever checked in the last baseline system. He or she is responsible for repairing the problem.
3. Make the changes to the system components.
4. Build the system in a private workspace and rerun system tests. If the tests fail, continue editing.
5. Once the system has passed its tests, check it into the build system server but do not commit it as a new system baseline in the VC system.
6. Build the system on the build server and run the tests. Alternatively, if you are using Git, you can pull recent changes from the server to your private workspace. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
7. If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

- Tools such as Jenkins are used to support continuous integration.
- These tools can be set up to build a system as soon as a developer has completed a repository update.
- The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon aspossible.
- The most recent system in the mainline is the definitive working system.
- However, although continuous integration is a good idea, it is not always possible toimplement this approach to system building:

1. If the system is very large, it may take a long time to build and test, especially if integration withother application systems is involved. It may be impractical to build the system being developedseveral times per day.

   If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace. There may be differences in hardware, operating system, or installed software. Therefore, more time is required for testing the system

- For large systems or for systems where the execution platform is not the same as the development platform, continuous integration is usually impossible. In those circumstances, frequent system building is supported using a **daily build system**:

1. The development organization sets a delivery time (say 2 p.m.) for system components. If developershave new versions of the components that they are writing, they must deliver them by that time. Components may be incomplete but should provide some basic functionality that can be tested.
2. A new version of the system is built from these components by compiling and linking them to form acomplete system.
3. This system is then delivered to the testing team, which carries out a set of predefined system tests.
4. Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

The advantages of using frequent builds of software are that the chances of finding problems stemming from component interactions early in the process are increased. Frequent building encourages thorough unit testingof components.

Frequent building encourages thorough unit testing of components.

- As compilation is a computationally intensive process, tools to support system building may be designed to minimize the amount of compilation that is required. They do this bychecking if a compiled version of a component is available. If so, there is no need to recompile that component.
- Therefore, there has to be a way of unambiguously linking the source code of acomponent with its equivalent object code.
- This linking is accomplished by associating a unique signature with each file where a source code component is stored.

- The corresponding object code, which has been compiled from the source code, has arelated signature.
- The signature identifies each source code version and is changed when the source code is edited. By comparing the signatures on the source and object code files, it is possible to decide if the source code component was used to generate the object code component
- Two types of signature may be used.(figure 10)

1. Modification timestamps:

- The signature on the source code file is the time and date when that file was modified.
- If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.
- [For example, say components Comp.java and Comp.class have modification signatures of 17:03:05:02:14:2014 and 16:58:43:02:14:2014, respectively. This means that the Java code was modified at 3 minutes and 5 seconds past 5 on the 14th of February 2014 and the compiled versionwas modified at 58 minutes and 43 seconds past 4 on the 14th of February 2014. In this case, the system would automatically recompile Comp.java because the compiled version has an earlier modification date than the most recent version of the component.]

2. Source code checksums

- The signature on the source code file is a checksum calculated from data in the file.
- A checksum function calculates a unique number using the source text as input.
- If you change the source code (even by one character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.
- The checksum is assigned to the source code just before compilation and uniquely identifies the source file.
- The build system then tags the generated object code file with the checksum signature.
- If there is no object code file with the same signature as the source code file to be included in a system, then recompilation of the source code isnecessary

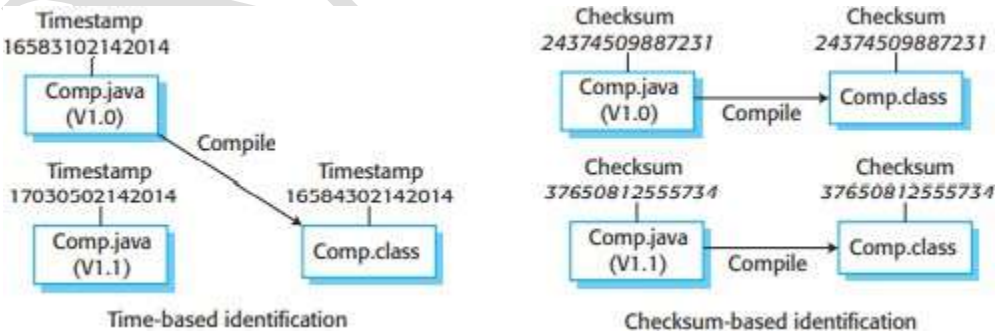
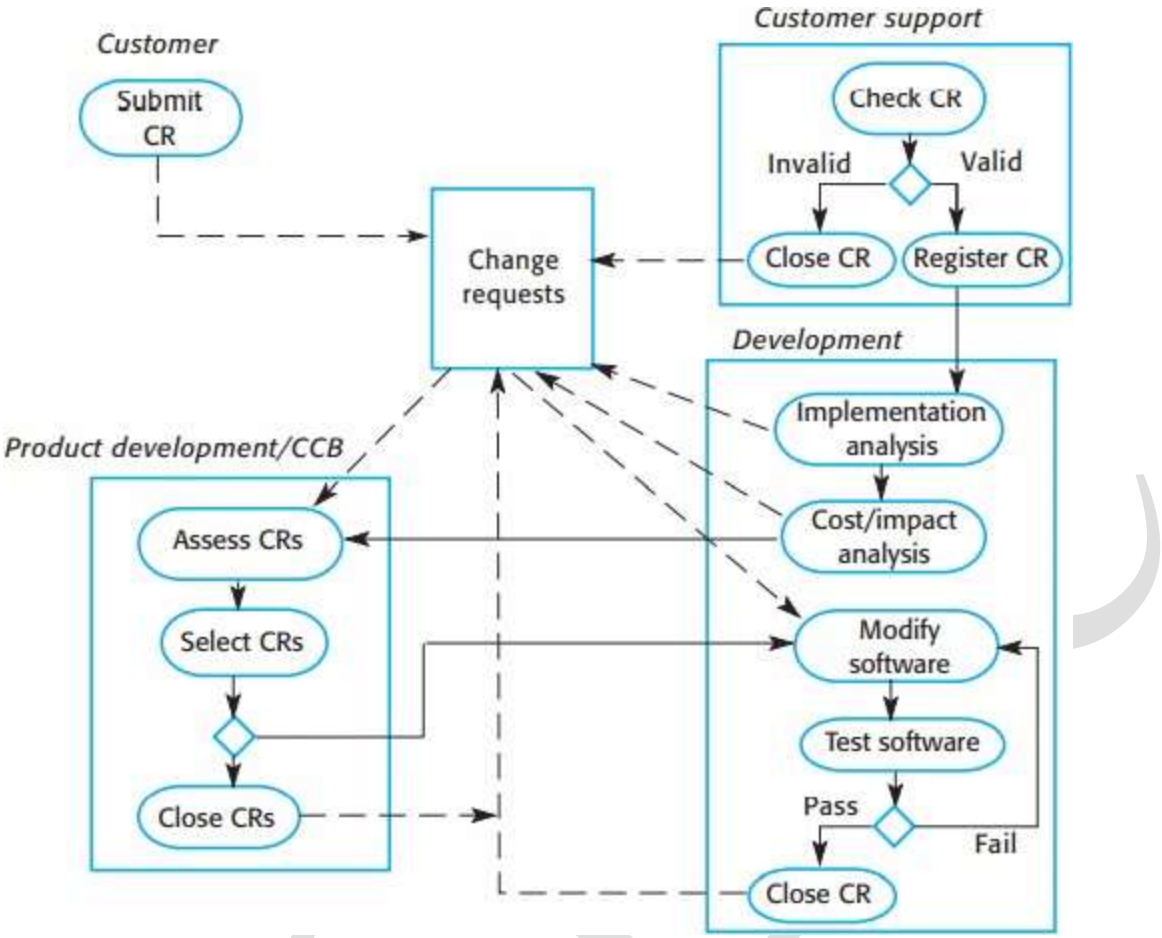


Figure 10: Linking source and object code

- As object code files are not normally versioned, the first approach(modification timestamps) means that only the most recently compiled object code file is maintained in the system.
- This is normally related to the source code file by name; that is, it has the same name as the source code filebut with a different suffix. Therefore, the source file Comp.Java may generate the object file Comp.class.
- Because source and object files are linked by name, it is not usually possible to build different versions of asource code component into the same directory at the same time.
- The compiler would generate object files with the same name, so only the most recently compiled versionwould be available.
- The checksum approach has the advantage of allowing many different versions of the object code of acomponent to be maintained at the same time.
- The signature rather than the filename is the link between source and object code. The source code andobject code files have the same signature. Therefore, when you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used.
- Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible, and different versions of a component may be compiled at the same time.
- Change management
  Change is a fact of life for large software systems. Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired,and systems have to adapt to changes in their environment.
- To ensure that the changes are applied to the system in a controlled way, youneed a set of tool-supported, change management processes.
- **Change management is intended to ensure that the evolution of the system is controlled and that the most urgent and cost-effective changes are prioritized.**
- **Change management is the process of analyzing the costs and benefits ofproposed changes, approving those changes that are cost-effective, and tracking which components in the system have been changed.**
- Figure 11 is a model of a change management process that shows the main change management activities. This process should come into effect when thesoftware is handed over for release to customers or for deployment within anorganization



- Many variants of this process are in use depending on whether the software is a custom system, a product line, or an off-the-shelf product. The size of the company also makes adifference—small companies use a less formal process than large companies that are working with corporate or government customers.]
- All change management processes should include some way of checking, costing, andapproving changes.
- Tools to support change management may be relatively simple issue or bug trackingsystems or software that is integrated with a configuration management package forlarge-scale systems, such as Rational Clearcase.
- Issue tracking systems allow anyone to report a bug or make a suggestion for a system change, and they keep track of how the development team has responded to the issues.
- More complex systems are built around a process model of the change management process. They automate the entire process of handling change requests from the initial customer proposal to final change approval and change submission to the developmentteam.

- ***The change management process is initiated when a system stakeholder completesand submits a change request describing the change required to the system. member of the development team. The change request may be rejected at this stage.***

- If the change request is a bug report, the bug may have already been reported and repaired.
- Sometimes, what people believe to be problems are actually misunderstandings of what the system isexpected to do.
- On occasions, people request features that have already been implemented but that they don't know about.
  - **If any of these features are true(ie. the change is not valid), the issue is closed and the form is updatedwith the reason for closure**.
  - **If it is a valid change request, it is then logged as an outstanding request for subsequent analysis**.
  - **For valid change requests, the next stage of the process is change assessment and costing**.
- This function is usually the responsibility of the development or maintenance team as they can work out what is involved in implementing the change
  - The impact of the change on the rest of the system must be checked. To do this, you have to identify all of the componentsaffected by the change.
  - If making the change means that further changes elsewhere in the system are needed, this will obviously increase the cost ofchange implementation.
  - Next, the required changes to the system modules are assessed.
  - Finally, the cost of making the change is estimated, taking into account the costs of changing related components
  - Following this analysis, **a separate group decides if it is cost-effective for the business to make the change to the software**.
- For military and government systems, this group is often called the **change control board (CCB**).
  - In industry, it may be called something like a "**product development group**" responsible for making decisions about how a software system should evolve.
  - This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, web pages, or documents.
  - These small requests should be passed to the development team for immediate implementation. The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view.
  - It decides whether the change in question is economically justified, and it prioritizes accepted changes for implementation.
  - **Accepted changes are passed back to the development group**; **rejected change requests are closed and no further action istaken**.
- The consequences of not making the change: When assessing a change request, you have to consider what will happen if the change is not implemented.[If the change is associated with a reported system failure, the seriousness of that failure has to be taken into account. If the system failure causes the systemto crash, this is very serious, and failure to make the change may disrupt the operational use of the system. On the other hand, if the failure has a minor effect, such as incorrect colors on a display, then it is not important to fix the problem quickly. The change should therefore have a low priority.]

- The benefits of the change: Will the change benefit many users of the system, or will it only benefit thechange proposer?
- The number of users affected by the change: If only a few users are affected, then the change may be assigned a low priority. In fact, making the change may be inadvisable if it means that the majority ofsystem users have to adapt to it.
- The costs of making the change If making the change affects many system components (hence increasing the chances of introducing new bugs) and/or takes a lot of time to implement, then the change may be rejected.
- The product release cycle If a new version of the software has just been released to customers, it may make sense to delay implementation of the change until the next planned release

- **Change management for software products (e.g., a CAD system product), rather than custom systems specifically developed for a certain customer, are handled in a different way.**
- In software products, the customer is not directly involved in decisions about system evolution, so therelevance of the change to the customer's business is not an issue.
- Change requests for these products come from the customer support team, the company marketing team,and the developers themselves. These requests may reflect suggestions and feedback from customers or analyses of what is offered by competing products.
- The customer support team may submit change requests associated with bugs that have been discovered and reported by customers after the software has been released.
- Customers may use a web page or email to report bugs. A bug management team then checks that the bug

reports are valid and translates them into formal system change requests.
- Marketing staff may meet with customers and investigate competitive products.
- They may suggest changes that should be included to make it easier to sell a new version of a system to newand existing customers.
- The system developers themselves may have some good ideas about new features that can be added to thesystem.
- During development, when new versions of the system are created through daily (or more frequent) system builds, there is no need for a formal change management process.
- Problems and requested changes are recorded in an issue tracking system and discussed in daily meetings.

- Changes that only affect individual components are passed directly to the system developer, who either accepts them or makes a case for why they

are not required.
- However, an independent authority, such as the system architect, should assess and prioritize changes that cut across system modules that have beenproduced by different development teams.
- In some agile methods, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the

change should take priority over the features planned for the next increment of the system.

- However, changes that involve software improvement are left to the discretion of the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.
- **As the development team changes software components, they should maintain a record of the changes made to each component. This is sometimes called the derivation history of a component**.
- A good way to keep the derivation history is in a **standardized comment at the beginning of the component source** code (Figure 12). This comment should reference the change request that triggered the software change. These comments can be processed by scripts that scan all components forthe derivation histories and then generate component change reports.

- For documents, records of changes incorporated in each version are usually maintained in a separate page at the front of the document.

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2012
//
// © St Andrews University 2012
//
// Modification history
// Version   Modifier   Date         Change      Reason
// 1.0       J. Jones   11/11/2009   Add header  Submitted to CM
// 1.1       R. Looek   13/11/2009   New field   Change req. R07/02
```

12: Derivation History

Release management

- A system release is a version of a software system that is distributed to customers.
- For mass-market software, it is usually possible to identify two types of release: **major releases** which deliver significant new functionality, and **minor releases**, which repairbugs and fix customer problems that have been reported.
- A software product release is not just the executable code of the system.
- The release may also include: configuration files defining how the release should be configured for particular installations; data files, such as files of error messages in different languages, that are needed for successful system operation;
- an installation program that is used to help install the system on target hardware;
- electronic and paper documentation describing the system;packaging and associated publicity that have been designed for that release

- Preparing and distributing a **system release for mass-market products is an expensive process**.
- In addition to the technical work involved in creating a release distribution, advertising and publicity material have to be prepared.
- Marketing strategies may have to be designed to convince customers to buy the new release of the system.
- Careful thought must be given to **release timing**.
- If releases are too frequent or require hardware upgrades, customers maynot move to the new release, especially if they have to pay for it.
- If system releases are infrequent, market share may be lost as customers move to alternative systems.
- The various technical and organizational factors that you should takeinto account when deciding on when to release a new version of a software product are shown in Figure 13.

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date. For marketing reasons, it may be necessary to include new features in a system so that users can be persuaded to upgrade from a previous release.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported that affect the way in which many customers use the system, it may be necessary to correct them in a new system release. Minor system faults may be repaired by issuing patches, distributed over the Internet, which can be applied to the current release of the system.

Figure 13: Factors influencing system release planning

- ***Release creation is the process of creating the collection of files and documentation that include all components of the system release.***

This process involves several steps:

- The executable code of the programs and all associated data files must be identified in the version control system and tagged with the release identifier.
- Configuration descriptions may have to be written for different hardware andoperating systems.
- Updated instructions may have to be written for customers who need to configure their own systems.
- Scripts for the installation program may have to be written.
- Web pages have to be created describing the release, with links to systemdocumentation.
- Finally, when all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

- For custom software or software product lines, the complexity of the system releasemanagement process depends on the number of system customers.
- Special releases of the system may have to be produced for each customer.
- Individual customers may be running several different releases of the system at the same time on different hardware.
- Where the software is part of a complex system of systems, different variants of theindividual systems may have to be created.
- A software company may have to manage tens or even hundreds of different releases of their software.
- Their configuration management systems and processes have to be designed to provide information about which customers have which releases of the system and the relationship between releases and system versions.
- In the event of a problem with a delivered system, you have to be able to recover all of the component versions used in that specific system
- When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- This is particularly important for customized, long-lifetime embedded systems, such as military systems and those that control complex machines. These systems may have a long lifetime—30 years in some cases.
- Customers may use a single release of these systems for many years and may require specific changes to that release long after it has been superseded.
- To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- You must keep copies of the source code files, corresponding executables, and all data and configuration files.
- It may be necessary to keep copies of older operating systems and other supportsoftware because they may still be in operational use.
- You should also record the versions of the operating system, libraries, compilers, and other tools used to build the software.
- These tools may be required in order to build exactly the same systemat some later date.
- Accordingly, you may have to store copies of the platform software and the tools used to create the system in the version control system,along with the source code of the target system.
- When planning the installation of new system releases, you cannot assume that customers will always install new system releases. Some system users may be happy anexisting system and may not consider it worthwhile to absorb the cost of changing to anew release.
- New releases of the system cannot, therefore, rely on the installation of previousreleases.
- One benefit of delivering software as a service (SaaS) is that it avoids all of theseproblems.

- It simplifies both release management and system installation for customers.
- The software developer is responsible for replacing the existing release of a system with a new release, which is made available to all customers at the same time.
- However, this approach requires that all servers running the services be updated at the same time. To support server updates, specialized distribution management tools such as Puppet have been developed for "pushing" new software to servers.

MODULE 5 NOTES

SOFTWARE QUALITY, PROCESS IMPROVEMENT AND TECHNOLOGY TRENDS

Today, software quality remains an issue, but who is to blame? Customers blame developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated. Who's right? *Both* —and that's the problem.

What is Quality?

Quality can be described from five different points of view.

(1) The ***transcendental view*** argues that quality is something you immediately recognize, but cannot explicitly define.

(2) The ***user view*** sees quality in terms of an end user's specific goals. If a product meets those goals, it exhibits quality.

(3) The ***manufacturer's view*** defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality.

(4) The ***product view*** suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.

(5) The ***value-based view*** measures quality based on how much a customer is willing to pay for a product.

In reality, quality encompasses all of these views and more.

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases if the product is manufactured according to specifications. In software development, quality of design encompasses the degree to which the design meets the functions and features specified in the requirements model. ***Quality of conformance*** focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

User satisfaction = compliant product + good quality + delivery within budget and schedule

—A product's quality is a function of how much it changes the world for the better.‖

SOFTWARE QUALITY

Software quality can be defined as: An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.

Garvin's Quality Dimensions

There are 8 dimensions of quality:

(1) **Performance Quality.** Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end user?

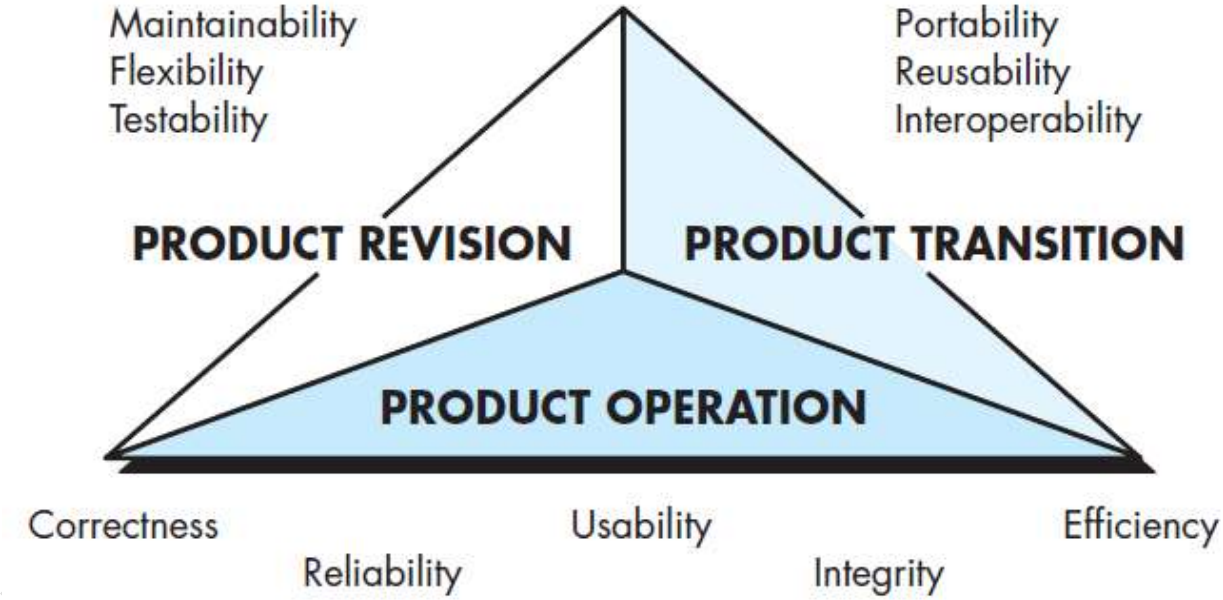
(2) **Feature quality.** Does the software provide features that surprise and delight first-time end users?

(3) **Reliability.** Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?

(4) **Conformance.** Does the software conform to local and external software standards that are relevant to the application?

(5) **Durability.** Will changes cause the error rate or reliability to degrade with time?
(6) **Serviceability.** Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period?
(7) **Aesthetics.** Each of us has a different and very subjective vision of what is aesthetic. An aesthetic entity has certain elegance, a unique flow, and an obvious ―presence‖ that are hard to quantify but are evident nonetheless. Aesthetic software has these characteristics.
(8) **Perception.** In some situations, you have a set of prejudices that will influence your perception of quality. For example, if you are introduced to a software product that was built by a vendor who has produced poor quality in the past, your guard will be raised and your perception of the current software product quality might be influenced negatively. Similarly, if a vendor has an excellent reputation, you may perceive quality, even when it does not really exist.

McCall's Quality Factors



- *Reliability.* The extent to which a program can be expected to perform its intended function with required precision.
- *Efficiency.* The amount of computing resources and code required by a program to perform its function.
- *Integrity.* Extent to which access to software or data by unauthorized persons can be controlled.
- *Usability.* Effort required to learn, operate, prepare input for, and interpret output of a program.
- *Maintainability.* Effort required to locate and fix an error in a program.

- *Flexibility.* Effort required to modify an operational program.
- *Testability.* Effort required to test a program to ensure that it performs its intended function.
- *Portability.* Effort required to transfer the program from one hardware and/or software system environment to another.
- *Reusability.* Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.
- *Interoperability.* Effort required to couple one system to another.

ISO 9126 Quality Factors

ISO 9126 standard identifies six key quality attributes:
(1) **Functionality.** The degree to which the software satisfies stated needs as indicated by the following sub attributes: suitability, accuracy, interoperability, compliance, and security.
(2) **Reliability.** The amount of time that the software is available for use as indicated by the following sub attributes: maturity, fault tolerance, recoverability.
(3) **Usability.** The degree to which the software is easy to use as indicated by the following sub attributes: understandability, learnability, operability.
(4) **Efficiency.** The degree to which the software makes optimal use of system resources as indicated by the following sub attributes: time behavior, resource behavior.
(5) **Maintainability.** The ease with which repair may be made to the software as indicated by the following sub attributes: analyzability, changeability, stability, testability.
(6) **Portability.** The ease with which the software can be transposed from one environment to another as indicated by the following sub attributes: adaptability, install ability, conformance, replace ability.

THE SOFTWARE QUALITY DILEMMA

If you produce a software system that has terrible quality, you lose because no one will want to buy it. If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it‘s going to take so long to complete and it will be so expensive to produce that you‘ll be out of business anyway. So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away.

“Good Enough” Software:

What is ―good enough‖? Good enough software delivers high-quality functions and features that end users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs. The software vendor hopes that the vast majority of end users will overlook the bugs because they are so happy with other application functionality.

The Cost of Quality:
We know that quality is important, but it costs us time and money—too much time and money to get the level of software quality we really want. The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

- ***Prevention costs*** include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities.
- ***Appraisal costs*** include activities to gain insight into product condition the ―first time through‖ each process. Examples of appraisal costs include: (1) the cost of conducting technical reviews for software engineering work products, (2) the cost of data collection and metrics evaluation, and (3) the cost of testing and debugging
- ***Failure costs*** are those that would disappear if no errors appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. ***Internal failure costs*** are incurred when you detect an error in a product prior to shipment. Internal failure costs include: (1) the cost required to perform rework (repair) to correct an error, (2) the cost that occurs when rework inadvertently generates side effects that must be mitigated, and (3) the costs associated with the collection of quality metrics that allow an organization to assess the modes of failure. ***External failure costs*** are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labour costs associated with warranty work.

Risks:

Low-quality software increases risks for both the developer and the end user.

Negligence and Liability:
Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad. The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval. In most cases, the customer claims that the developer has been negligent and is therefore not entitled to payment. The developer often claims that the customer has repeatedly changed its requirements and has subverted the development partnership in other ways. In every case, the quality of the delivered system comes into question.
Quality and Security:
As the criticality of Web-based and mobile systems grows, application security has become increasingly important. To build a secure system, you must focus on quality, and that focus must begin during design.

The Impact of Management Actions:
As each project task is initiated, a project leader will make decisions that can have a significant impact on product quality.

- **Estimation decisions.** A software team is rarely given the luxury of providing an estimate for a project *before* delivery dates are established and an overall budget is specified. Instead, the team conducts a ―sanity check‖ to ensure that delivery dates and milestones are rational. As a consequence, shortcuts are taken, activities that lead to higher-quality software may be skipped, and product quality suffers. If a delivery date is irrational, it is important to hold your ground. Explain why you need more time, or alternatively, suggest a subset of functionality that can be delivered (with high quality) in the time allotted.

- **Scheduling decisions.** When a software project schedule is established, tasks are sequenced based on dependencies. For example, because component **A** depends on processing that occurs within components B, C, and D, component A cannot be scheduled for testing until components B, C, and D are fully tested. A project schedule would reflect this. But if time is very short, and **A** must be available for further critical testing, you might decide to test **A** without its subordinate components (which are running slightly behind schedule), so that you can make it available for other testing that must be done before delivery. After all, the deadline looms. As a consequence, **A** may have defects that are hidden, only to be discovered much later. Quality suffers.
- **Risk-oriented decisions.** Risk management is one of the key attributes of a successful software project.

ACHIEVING SOFTWARE QUALITY

Software quality doesn‘t just appear. It is the result of good project management and solid software engineering practice. Management and practice are applied within the context of four broad activities that help a software team achieve high software quality:

1. Software engineering methods
2. Project management techniques
3. Quality control actions
4. Software quality assurance.

Software Engineering Methods
If you expect to build high-quality software, you must understand the problem to be solved. You must also be capable of creating a design that conforms to the problem while at the same time exhibiting characteristics that lead to software that exhibits the quality dimensions. There are a wide array of concepts and methods for that. If you apply those concepts and adopt appropriate analysis and design methods, the likelihood of creating high-quality software will increase substantially.

Project Management Techniques

Poor management decisions can impact the quality of the project. Software quality can be improved if:

(1) a project manager uses estimation to verify that delivery dates are achievable

(2) schedule dependencies are understood and the team resists the temptation to use shortcuts

(3) risk planning is conducted

- **Quality Control**

  Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. A combination of measurement and feedback allows a software team to tune the process when any of these work products fail to meet quality goals.

- **Quality Assurance**

  Quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions. The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality, thereby gaining insight and confidence that actions to achieve product quality are working.

ELEMENTS OF SOFTWARE QUALITY ASSURANCE

Software quality assurance (SQA) is often known as Quality Management. It encompasses a broad range of concerns and activities (also known as the elements of SQA).

Standards. The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers. Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

Testing. Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

Error/defect collection and analysis. SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

Change management. If change is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices have been instituted.

Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders.

Vendor management. Three categories of software are acquired from external software vendors— *shrink-wrapped packages* (e.g., Microsoft Office), a *tailored shell* that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and *contracted software* that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor.
Security management. SQA ensures that appropriate process and technology are used to achieve software security.

Safety. Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

Risk management. The SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

SQA TASKS (What is the role of SQA group?)
The Software Engineering Institute (SEI) recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that:

Prepares an SQA plan for a project. The plan is developed as part of project planning and is reviewed by all stakeholders. Quality assurance activities performed are governed by the plan. The plan identifies evaluations to be performed, audits and reviews to be conducted, standards that are applicable to the project, procedures for error reporting and tracking, work products that are produced by the SQA group, and feedback that will be provided to the software team.

Participates in the development of the project's software process description. The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Reviews software engineering activities to verify compliance with the defined software process. The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.

Records any noncompliance and reports to senior management.
Noncompliance items are tracked until they are resolved.
SQA Goals:

Requirements quality. The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality. Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality.

Code quality. Source code and related work products must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness. A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. SQA analyses the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

SOFTWARE PROCESS IMPROVEMENT (SPI)

The term *software process improvement* (SPI) implies many things. First, it implies that elements of an effective software process can be defined in an effective manner; second, that an existing organizational approach to software development can be assessed against those elements; and third, that a meaningful strategy for improvement can be defined. In short, SPI implies a defined software process, an organizational approach, and a strategy for improvement. The SPI strategy transforms the existing approach to software development into something that is more focused, more repeatable, and more reliable (in terms of the quality of the product produced and the timeliness of delivery).

Approaches to SPI

An organization can choose one of the many SPI frameworks. An ***SPI framework*** defines:
1. a set of characteristics that must be present if an effective software process is to be achieved
2. a method for assessing whether those characteristics are present
3. a mechanism for summarizing the results of any assessment
4. a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.

An SPI framework assesses the ―maturity‖ of an organization‘s software process and provides a qualitative indication of a maturity level.

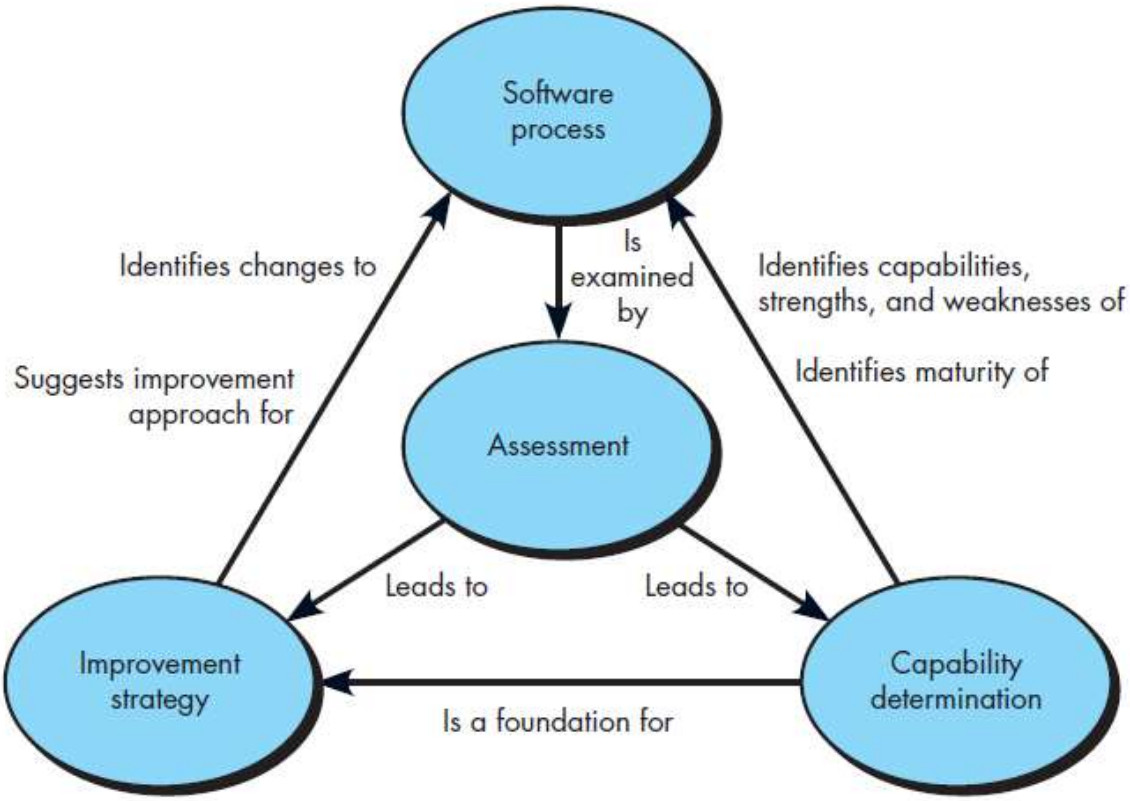


Fig: Elements of a SPI framework

There is no universal SPI framework. The SPI framework that is chosen by an organization reflects the constituency that is championing the SPI effort.
There are six different SPI support constituencies:

1. Quality certifiers: Process improvement efforts championed by this group focus on the following relationship:

Quality (*Process*) ⇒ Quality (*Product*)

Their approach is to emphasize assessment methods and to examine a well-defined set of characteristics that allows them to determine whether the process exhibits quality. They are most likely to adopt a process framework such as the CMMI, SPICE, TickIT, or Bootstrap.

2. Formalists. This group wants to understand process workflow. To accomplish this, they use process modeling languages (PMLs) to create a model of the existing process and then design extensions or modifications that will make the process more effective.

3. Tool advocates. This group insists on a tool-assisted approach to SPI

4. Practitioners. This constituency uses a **pragmatic approach, ―**emphasizing mainstream project-, quality- and product management, applying project-level planning and metrics, but with little formal process modeling or enactment support‖

5. Reformers. The goal of this group is organizational change that might lead to a better software process. They tend to focus more on human issues and emphasize measures of human capability.

6. Ideologists. This group focuses on the suitability of a particular process model for a specific application domain or organizational structure. Rather than typical software process models (e.g., iterative models), ideologists would have a greater interest in a process that would support reuse or reengineering.

Maturity Models

A *maturity model* is applied within the context of an SPI framework. The intent of the maturity model is to provide an overall indication of the ―process maturity‖ exhibited by a software organization. That is, an indication of the quality of the software process, the degree to which practitioners understand and apply the process.
There are four levels of organizational Immaturity.
Level 0, ***Negligent*** ―Failure to allow successful development process to succeed.
Level 1, ***Obstructive*** ―Counterproductive processes are imposed.
Level 2, ***Contemptuous*** ―Disregard for good software engineering practices, separation between software development activities and software process improvement activities, lack of a training program.
Level 3, ***Undermining*** ―Total neglect of own charter, conscious discrediting of peer organizations software process improvement efforts. These activities reward failure and poor performance.

THE SPI PROCESS

The Software Engineering Institute has developed IDEAL— ―an organizational improvement model that serves as a road map for SPI activities. There are five SPI activities:

1. Assessment and Gap analysis
2. Education and Training
3. Selection and Justification
4. Installation/Migration
5. Evaluation

1. Assessment and Gap analysis

Assessment: The intent of assessment is to uncover both strengths and weaknesses in the way your organization applies the existing software process and the software engineering practices that populate the process. Assessment examines a wide range of actions and tasks that will lead to a high-quality process. As the process assessment is conducted, you should also focus on the following issues:

- **Consistency.** Are important activities, actions, and tasks applied consistently across all software projects and by all software teams?
- **Sophistication.** Are management and technical actions performed with a level of sophistication that implies a thorough understanding of best practice?
- **Acceptance.** Is the software process and software engineering practice widely accepted by management and technical staff?
- **Commitment.** Has management committed the resources required to achieve consistency, sophistication, and acceptance?

Gap analysis: The difference between local application and best practice represents a ―gap‖ that offers opportunities for improvement.

2. Education and Training

A key element of any SPI strategy is education and training for practitioners, technical managers, and more senior managers. Three types of education and training should be conducted:

- Generic software engineering concepts and methods
- Specific technology and tools
- Communication and quality-oriented topics.

3. Selection and Justification

Once the initial assessment activity has been completed and education has begun, a software organization should begin to make choices.

- First, you should choose the process model that best fits your organization, its stakeholders, and the software that you build.
- You should decide which of the set of framework activities will be applied, the major work products that will be produced, and the quality assurance checkpoints that will enable your team to assess progress.
- Next, develop an adaptable work breakdown for each framework activity, defining the task set that would be applied for a typical project.

4. Installation/Migration

Installation is the first point at which a software organization feels the effects of changes implemented as a consequence of the SPI road map. In some cases, an entirely new process is recommended for an organization. In other cases, changes associated with SPI are relatively minor, representing small, but meaningful modifications to an existing process model. Such changes are often referred to as ***process migration***.
Installation and migration are actually ***software process redesign*** **(SPR)** activities. SPR is concerned with identification, application, and refinement of new ways to dramatically improve and transform software processes. When a formal approach to SPR is initiated, three different process models are considered:
(1) the existing (—as is‖) process
(2) a transitional (—here to there‖) process
(3) the target (—to be‖) process.

5. Evaluation

The evaluation activity assesses:

- the degree to which changes have been instantiated and adopted
- the degree to which such changes result in better software quality or other tangible process benefits
- the overall status of the process and the organizational culture as SPI activities proceed.

Both qualitative factors and quantitative metrics are considered during the evaluation activity. From a qualitative point of view, past management and practitioner attitudes about the software process can be compared to attitudes polled after installation of process changes. Quantitative metrics are collected from projects that have used the transitional or —to be‖ process and compared with similar metrics that were collected for projects that were conducted under the —as is‖ process.

Risk Management for SPI:

SPI is a risky undertaking. A software organization should manage risk at three key points in the SPI process:

1. Prior to the initiation of the SPI road map
2. During the execution of SPI activities (assessment, education, selection, installation)
3. During the evaluation activity that follows the instantiation of some process characteristic.

In general, the following categories can be identified for SPI risk factors:

- budget and cost
- content and deliverables
- culture
- maintenance of SPI deliverables
- mission and goals
- organizational management
- organizational stability
- process stakeholders
- schedule for SPI development, SPI development environment, SPI development process, SPI project management, and SPI staff.

Within each category, a number of generic risk factors can be identified. For example, some of the generic risk factors defined for the culture category are:

- Attitude toward change, based on prior efforts to change

- Ability of organization members to manage meetings effectively Experience with quality programs, level of success Using the risk factors and generic attributes as a guide, a risk can be developed to isolate those risks that warrant further management attention.

The CMMI

CMMI: *Capability Maturity Model Integration*
Or the SEI-CMM (Software Engineering Institute – Capability Maturity Integration)
The CMMI represents a process meta-model in two different ways:
1. as a ―continuous‖ model
2. as a ―staged‖ model.

The **continuous CMMI meta-model** describes a process in two dimensions as illustrated in below figure.

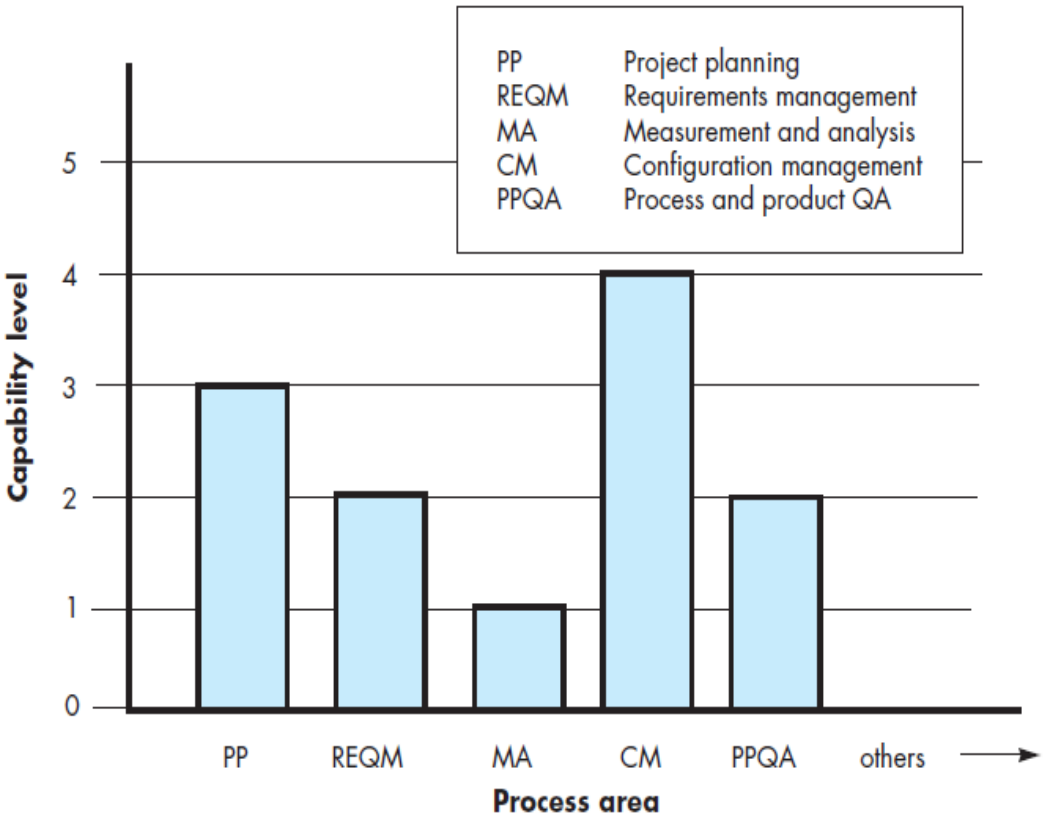


Fig: CMMI process capability profile

Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

- **Level 0:** ***Incomplete*** — The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.
- **Level 1:** ***Performed*** — All of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.
- **Level 2:** ***Managed*** — All capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are ―monitored, controlled, and reviewed; and are evaluated for adherence to the process description‖.
  Level 3: ***Defined*** — All capability level 2 criteria have been achieved. In addition, the process is ―tailored from the organization‘s set of standard processes‖
- **Level 4:** ***Quantitatively managed*** — All capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment.
- **Level 5:** ***Optimized*** — All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.
- The CMMI defines each process area in terms of ―specific goals‖ and the ―specific practices‖ required to achieve these goals. ***Specific goals*** establish the characteristics that must exist if the activities implied by a process area are to be effective. ***Specific practices*** refine a goal into a set of process-related activities.

For example, **project planning** is one of eight process areas defined by the CMMI for ―project management‖ category. The specific goals (SG) and the associated specific practices (SP) defined for **project planning** are:

SG 1 Establish Estimates
SP 1.1-1 Estimate the Scope of the Project
SP 1.2-1 Establish Estimates of Work Product and Task Attributes
SP 1.3-1 Define Project Life Cycle
SP 1.4-1 Determine Estimates of Effort and Cost
SG 2 Develop a Project Plan
SP 2.1-1 Establish the Budget and Schedule
SP 2.2-1 Identify Project Risks
SP 2.3-1 Plan for Data Management
SP 2.4-1 Plan for Project Resources
SP 2.5-1 Plan for Needed Knowledge and Skills
SP 2.6-1 Plan Stakeholder Involvement
SP 2.7-1 Establish the Project Plan

The **staged CMMI model** defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved.

THE ISO 9000 QUALITY STANDARDS (ISO 9001:2000 standard)

ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered. (ISO stands for ―*International Organization for Standardization").*
To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third party auditors for compliance to the standard and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semi-annual surveillance audits ensure continued compliance to the standard.

The requirements delineated by ISO 9001:2008

- Management responsibility
- Quality system
- Contract review
- Design control
- Document and data control
- Product identification and traceability
- Process control
- Inspection and testing
- Corrective and preventive action
- Control of quality records
- Internal quality audits
- Training, servicing, and statistical techniques.

INFO

The ISO 9001:2008 Standard

The following outline defines the basic elements of the ISO 9001:2000 standard. Comprehensive information on the standard can be obtained from the International Organization for Standardization (**www.iso.ch**) and other Internet sources (e.g., **www.praxiom.com**).

Establish the elements of a quality management system.
- Develop, implement, and improve the system.
- Define a policy that emphasizes the importance of the system.

Document the quality system.
- Describe the process.
- Produce an operational manual.
- Develop methods for controlling (updating) documents.
- Establish methods for record keeping.

Support quality control and assurance.
- Promote the importance of quality among all stakeholders.
- Focus on customer satisfaction.
- Define a quality plan that addresses objectives, responsibilities, and authority.
- Define communication mechanisms among stakeholders.

Establish review mechanisms for the quality management system.
- Identify review methods and feedback mechanisms.
- Define follow-up procedures.

Identify quality resources including personnel, training, and infrastructure elements.

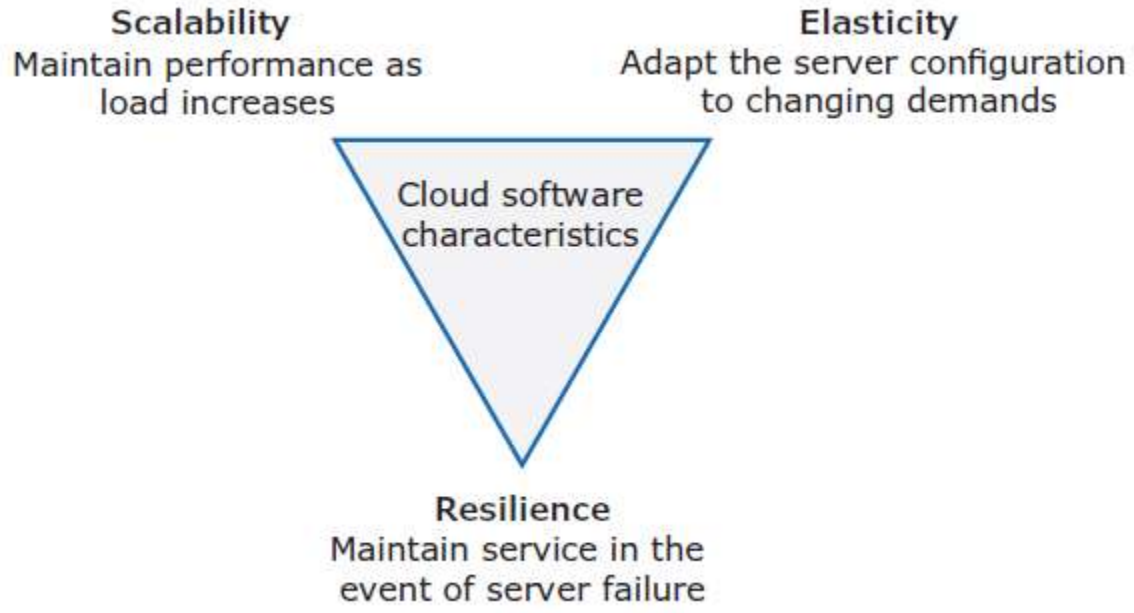
Establish control mechanisms.
- For planning.
- For customer requirements.
- For technical activities (e.g., analysis, design, testing).
- For project monitoring and management.

Define methods for remediation.
- Assess quality data and metrics.
- Define approach for continuous process and quality improvement.

Sr. No.	Key	ISO9000	SEI-CMM.
1	Definition	ISO9000 is an international standard of quality management and quality assurance. It certifies the companies that they are documenting the quality system elements which are needed to run a efficient and quality system.	SEI-CMM is specifically for software organizations to certify them at which level, they are following and maintaining the quality standards.
2	Focus	Focus of ISO9000 is on customer supplier relationship, and to reduce the customer's risk.	Focus of SEI-CMM is to improve the processes to deliver a quality software product to the customer.
3	Target Industry	ISO9000 is used by manufacturing industries.	SEI-CMM is used by software industry.
4	Recognition	ISO9000 is universally accepted across lots of countries.	SEI-CMM is mostly used in USA.
5	Guidelines	ISO9000 guides about concepts, principles and safeguards to be in place in a workplace.	SEI-CMM specifies what is to be followed at what level of maturity.
6	Levels	ISO9000 has one acceptance level.	SEI-CMM has five acceptance levels.
7	Validity	ISO9000 certificate is valid for three years.	SEI-CMM certificate is valid for three years as well.
8	Level	ISO9000 has no levels.	SEI-CMM has five levels, Initial, Repeatable, Defined, Managed and Optimized.

CLOUD-BASED SOFTWARE

Cloud is a very large number of remote servers that are offered for rent by companies that own these servers. You may rent a server and install your own software, or you may pay for access to software products that are available on the cloud.
The cloud servers that you rent can be started up and shut down as demand changes. This means that software that runs on the cloud can be scalable, elastic, and resilient (Figure below). These three factors (scalability, elasticity, and resilience) are the fundamental differences between cloud-based systems and those hosted on dedicated servers.



- **Scalability** reflects the ability of your software to cope with increasing numbers of users. As the load on your software increases, the software automatically adapts to maintain the system performance and response time. Systems can be scaled by adding new servers or by migrating to a more powerful server. If a more powerful server is used, this is called **scaling up**. If new servers of the same type are added, this is called **scaling out**.

- **Elasticity** is related to scalability but allows for **scaling down** as well as **scaling up**. That is, you can monitor the demand on your application and add or remove servers dynamically as the number of users changes.

- **Resilience** means that you can design your software architecture to tolerate server failures. You can make several copies of your software available concurrently. If one of these fails, the others continue to provide a service.

The **benefits of adopting cloud-based approach rather than buying your own servers for software development** are shown in the below table.

Factor	Benefit
Cost	You avoid the initial capital costs of hardware procurement.
Startup time	You don't have to wait for hardware to be delivered before you can start work. Using the cloud, you can have servers up and running in a few minutes.
Server choice	If you find that the servers you are renting are not powerful enough, you can upgrade to more powerful systems. You can add servers for short-term requirements, such as load testing.
Distributed development	If you have a distributed development team, working from different locations, all team members have the same development environment and can seamlessly share all information.

Virtualization and Containers

All cloud servers are virtual servers. A virtual server runs on an underlying physical computer and is made up of an operating system plus a set of software packages that provide the server functionality required. The general idea is that a virtual server is a stand-alone system that can run on any hardware in the cloud. This ―run anywhere‖ characteristic is possible because the virtual server has no external dependencies. An external dependency means you need some software, that you are not developing yourself. For example, if you are developing in Python, you need a Python compiler, a Python interpreter, various Python libraries, and so on.
Virtual machines (VMs) can be used to implement virtual servers (Figure below).

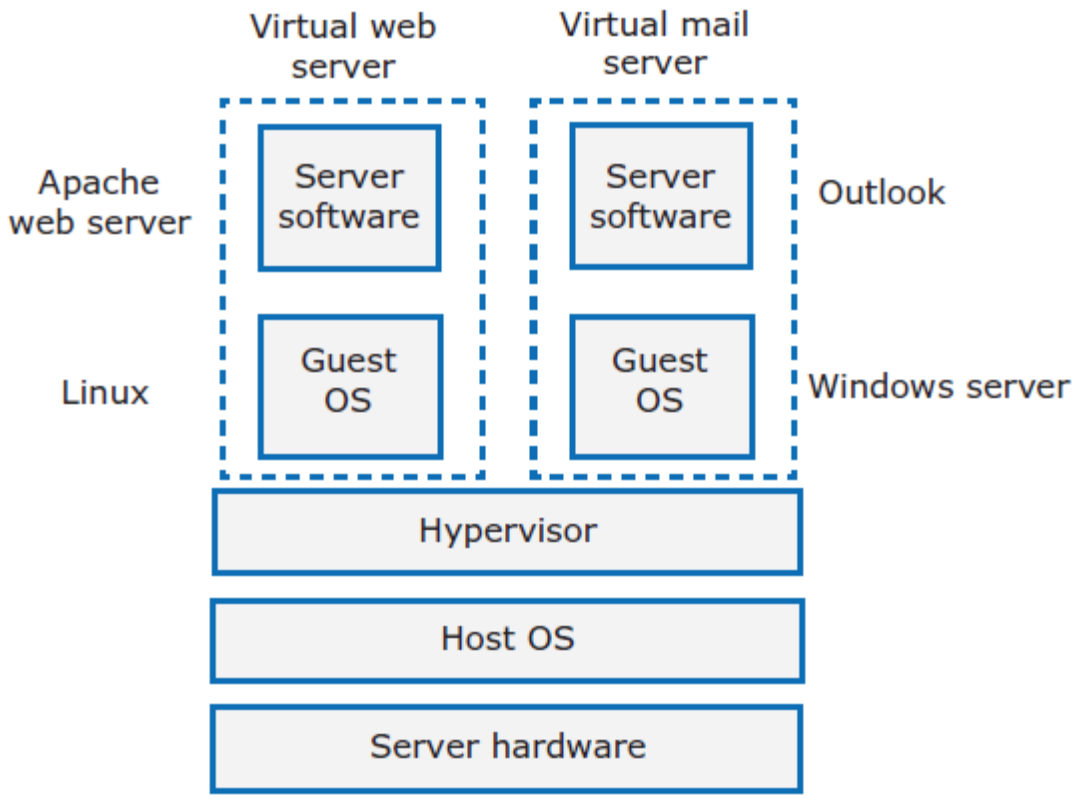


Fig: Implementing a virtual server as a virtual machine

Hypervisor provides a hardware emulation that simulates the operation of the underlying hardware. Several of these hardware emulators share the physical hardware and run in parallel. You can run an operating system and then install server software on each hardware emulator.

The advantage of using a virtual machine to implement virtual servers is that you have exactly the same hardware platform as a physical server. You can therefore run different operating systems on virtual machines that are hosted on the same computer. For example, the above figure shows that Linux and Windows can run concurrently on separate VMs.

Why software companies use container instead of virtual machines?

If you are running a cloud-based system with many instances of applications or services, these all use the same operating system, you can use a simpler virtualization technology called _containers‘.

Containers are an operating system virtualization technology that allows independent servers to share a single operating system. They are particularly useful for providing isolated application services where each user sees their own version of an application.

Using containers dramatically speeds up the process of deploying virtual servers on the cloud. Containers are usually megabytes in size, whereas VMs are gigabytes. Containers can be started up and shut down in a few seconds rather than the few minutes required for a VM. Many companies that provide cloud-based software have now switched from VMs to containers because containers are faster to load and less demanding of machine resources. These containers are managed by the means of Dockers.

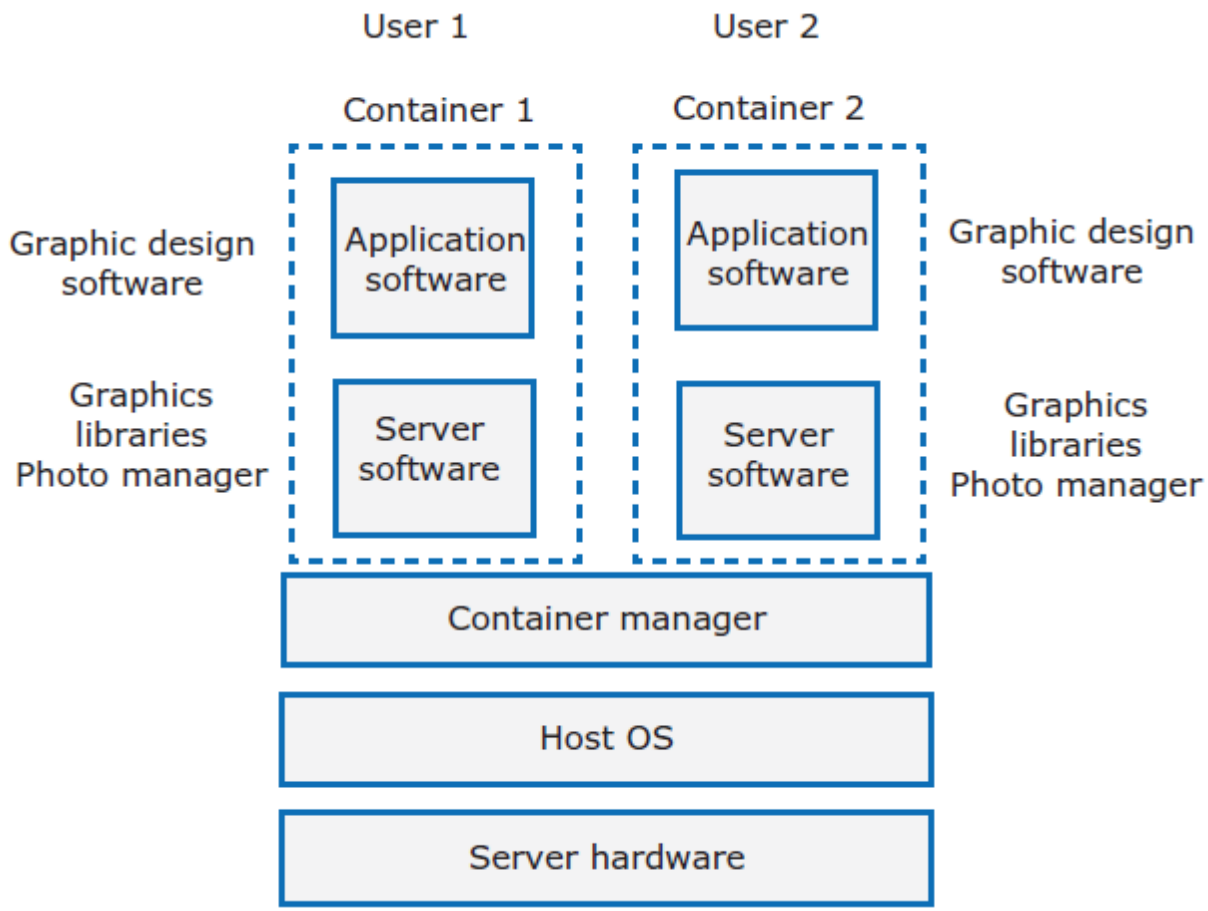


Fig: Using containers to provide isolated services

Docker: Docker is a container management system that allows users to define the software to be included in a container as a Docker image. It also includes a run-time system that can create and manage containers using these Docker images. Below figure shows the different elements of the Docker container system and their interactions. The function of each of the elements in the Docker container system is shown the table followed.

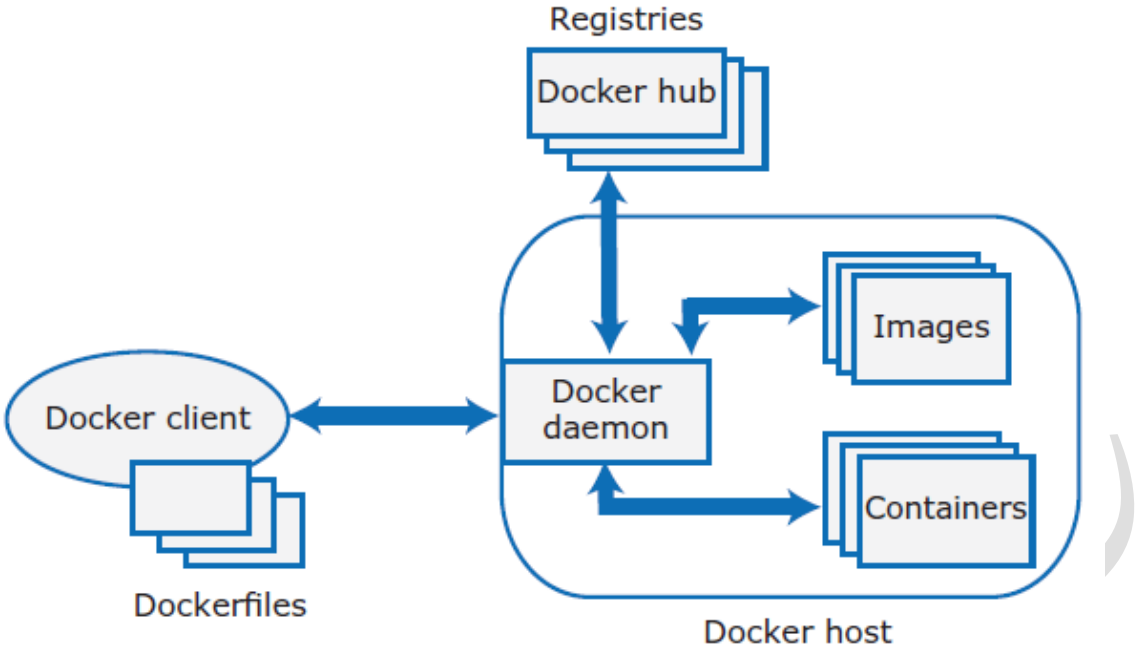


Fig: The Docker container system

Element	Function
Docker daemon	This is a process that runs on a host server and is used to set up, start, stop, and monitor containers, as well as building and managing local images.
Docker client	This software is used by developers and system managers to define and control containers.
Dockerfiles	Dockerfiles define runnable applications (images) as a series of setup commands that specify the software to be included in a container. Each container must be defined by an associated Dockerfile.
Image	A Dockerfile is interpreted to create a Docker image, which is a set of directories with the specified software and data installed in the right places. Images are set up to be runnable Docker applications.
Docker hub	This is a registry of images that has been created. These may be reused to set up containers or as a starting point for defining new images.
Containers	Containers are executing images. An image is loaded into a container and the application defined by the image starts execution. Containers may be moved from server to server without modification and replicated across many servers. You can make changes to a Docker container (e.g., by modifying files) but you then must commit these changes to create a new image and restart the container.

Table: The elements of the Docker container system

What are the benefits of containers?

1. They solve the problem of software dependencies.
2. You don‘t have to worry about the libraries and other software on the application server being different from those on your development server. Instead of shipping your product as stand-alone software, you can ship a container that includes all of the support software that your product needs.
3. They provide a mechanism for software portability across different clouds.
4. Docker containers can run on any system or cloud provider where the Docker daemon is available.
5. They provide an efficient mechanism for implementing software services and so support the development of service-oriented architectures.
6. They simplify the adoption of DevOps.

EVERYTHING AS A SERVICE

The idea of a service that is rented rather than owned is fundamental to cloud computing. Instead of owning hardware, you can rent the hardware that you need from a cloud provider. If you have a software product, you can use that rented hardware to deliver the product to your customers. In cloud computing, this has been developed into the idea of ―everything as a service.‖
There are three levels where everything as a service is most relevant:
1. Infrastructure as a service (IaaS)
2. Platform as a service (PaaS)
3. Software as a service (SaaS)

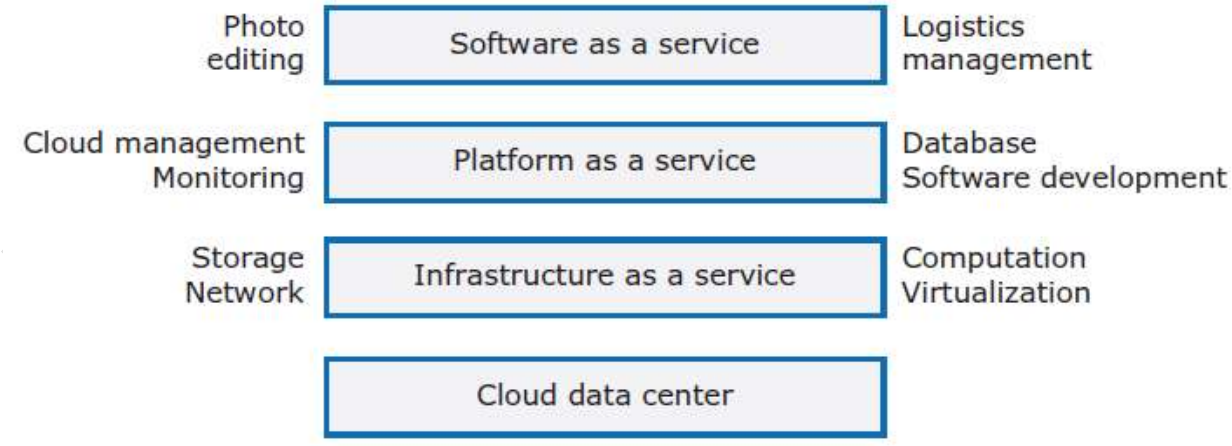


Fig: Everything as a service

Infrastructure as a service (IaaS) This is a basic service level that all major cloud providers offer. They provide different kinds of infrastructure service, such as a computer service, a network service, and a storage service. These infrastructure services may be used to implement virtual cloud-based servers.

The key **benefits** of using IaaS:

- You don‘t incur the capital costs of buying hardware
- You can easily migrate your software from one server to a more powerful server
- You can add more servers if you need to as the load on your system increases.

Platform as a service (PaaS) This is an intermediate level where you use libraries and frameworks provided by the cloud provider to implement your software. These provide access to a range of functions, including SQL and NoSQL databases. Using PaaS makes it easy to develop auto-scaling software. You can implement your product so that as the load increases, additional compute and storage resources are added automatically.

Software as a service (SaaS) Your software product runs on the cloud and is accessed by users through a web browser or mobile app. This type of cloud service includes—mail services such as Gmail, storage services such as Dropbox, social media services such as Twitter, and so on.

System management responsibilities of IaaS, PaaS and SaaS

- If you are using IaaS, you have the responsibility for installing and managing the database, the system security, and the application.
- If you use PaaS, you can devolve responsibility of managing the database and security to the cloud provider.
- In SaaS, assuming that a software vendor is running the system on a cloud, the software vendor manages the application. Everything else is the cloud provider‘s responsibility.

SOFTWARE AS A SERVICE

If you deliver your software product as a service, you run the software on your servers, which you may rent from a cloud provider. Customers don‘t have to install software, and they access the remote system through a web browser or dedicated mobile app (Figure below). The payment model for SaaS is usually a subscription. Users pay a monthly fee to use the software.

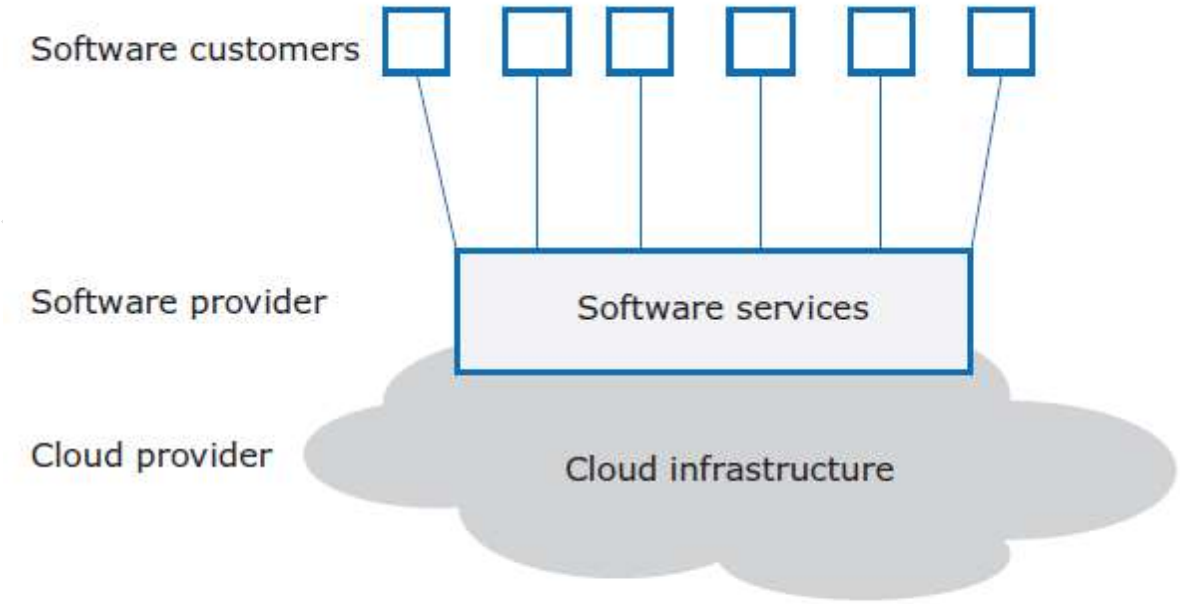


Fig: Software as a service

Many software providers deliver their software as a cloud service, but also allow users to download a version of the software so that they can work without a network connection. For example, Adobe offers the Lightroom photo management software as both a cloud service and a download that runs on the user‘s own computer. This gets around the problem of reduced performance due to slow network connections.

Benefits of SaaS for software product providers:

Benefit	Explanation
Cash flow	Customers either pay a regular subscription or pay as they use the software. This means you have a regular cash flow, with payments throughout the year. You don't have a situation where you have a large cash injection when products are purchased but very little income between product releases.
Update management	You are in control of updates to your product, and all customers receive the update at the same time. You avoid the issue of several versions being simultaneously used and maintained. This reduces your costs and makes it easier to maintain a consistent software code base.
Continuous deployment	You can deploy new versions of your software as soon as changes have been made and tested. This means you can fix bugs quickly so that your software reliability can continuously improve.
Payment flexibility	You can have several different payment options so that you can attract a wider range of customers. Small companies or individuals need not be discouraged by having to pay large upfront software costs.
Try before you buy	You can make early free or low-cost versions of the software available quickly with the aim of getting customer feedback on bugs and how the product could be approved.
Data collection	You can easily collect data on how the product is used and so identify areas for improvement. You may also be able to collect customer data that allow you to market other products to these customers.

Advantages (Benefits) and disadvantages of SaaS for customers:

One of the most significant business benefits of using SaaS is that customers do not incur the capital costs of buying servers or the software itself. However, customers have to continue to pay, even if they rarely use the software.

The universal use of mobile devices means that customers want to access software from these devices as well as from desktop and laptop computers. People can use the software from multiple devices without having to install the software in advance. However, this may mean that software developers have to develop mobile apps for a range of platforms in order to maintain their customer base.

A further benefit of SaaS for customers is that they don‘t have to employ staff to install and update the system. However, this may lead to a loss of local expertise. A lack of expertise may make it more difficult for customers to revert to self-hosted software if they need to do so.
A characteristic of SaaS is that updates can be delivered quickly. New features are immediately available to all customers. However, customers have no control over when software upgrades are installed.
Other disadvantages of SaaS are related to storage and data management issues. These issues are given in the below table.

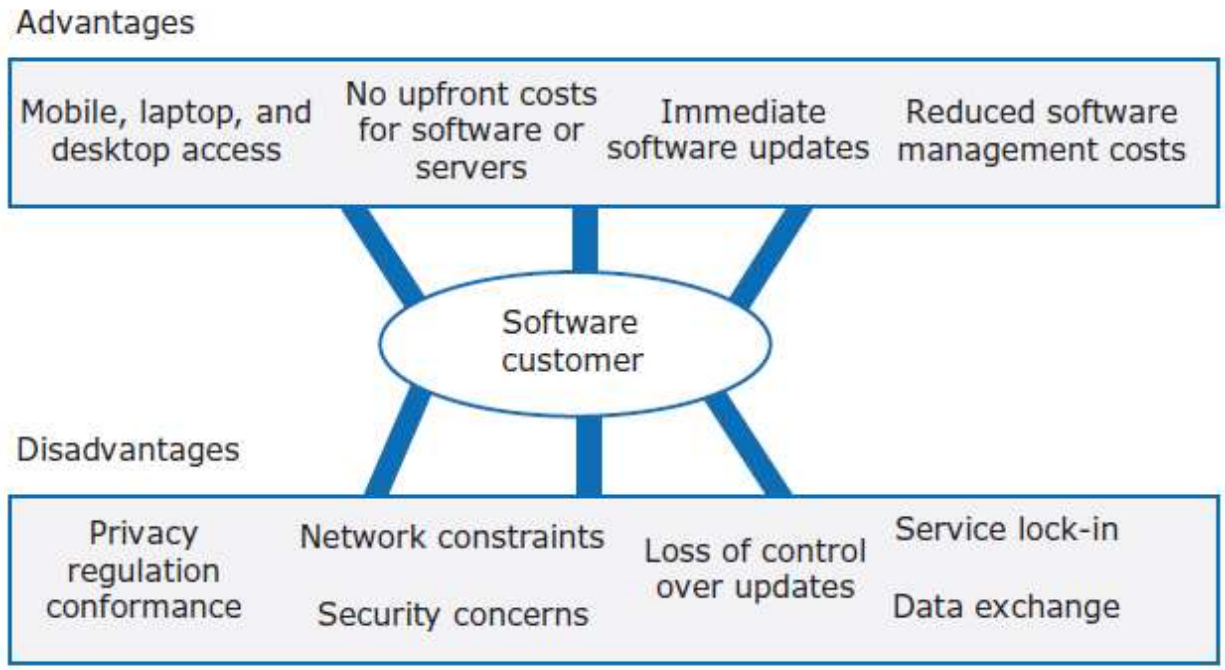


Table: Data storage and management issues for SaaS

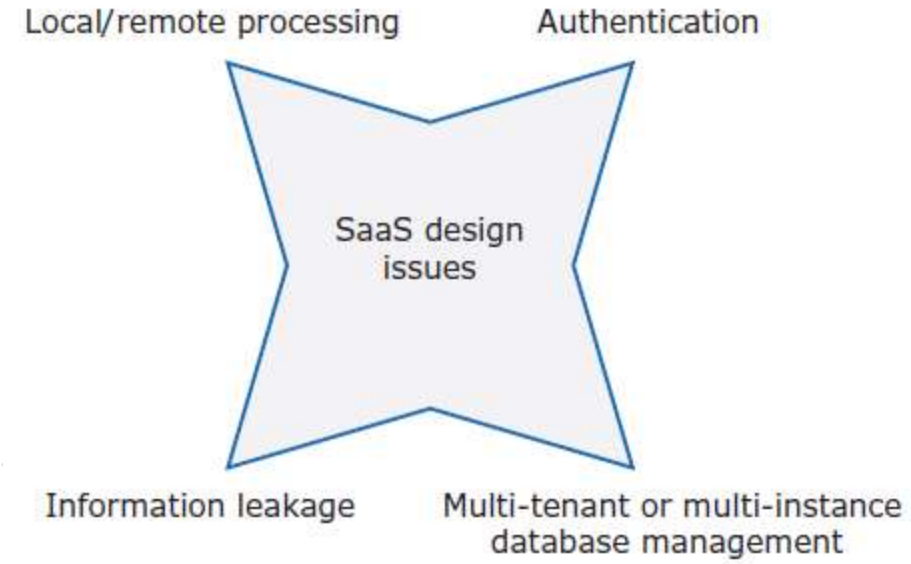
Design issues for SaaS: (The factors that you have to consider while designing the software)

A software product may be designed so that some features are executed locally in the user‘s browser or mobile app and some on a remote server. Local execution reduces network traffic and so increases user response speed. This is useful when users have a slow network connection.

On all shared systems, users have to authenticate themselves to show that they are accredited to use the system. You can set up your own authentication system, but this means users have to remember another set of authentication credentials. People don‘t like this, so for individual users, many systems allow authentication using the user‘s Google, Facebook, or LinkedIn credentials.

Information leakage is a particular risk for cloud-based software. If you have multiple users from multiple organizations, a security risk is that information leaks from one organization to another. So you need to be very careful in designing your security system to avoid it.

Multi-tenancy means that the system maintains the information from different organizations in a single repository rather than maintaining separate copies of the system and database. This can lead to more efficient operation. However, the developer has to design software so that each organization sees a virtual system that includes its own configuration and data. In a multi-instance system, each customer has their own instance of the software and its database.



MICROSERVICES ARCHITECTURE

(Microservices, Microservice Architecture, Microservice Deployment)

Software Service:

A software service is a software component that can be accessed from remote computers over the Internet. Given an input, a service produces a corresponding output, without side effects. The service is accessed through its published interface and all details of the service

implementation are hidden. The manager of a service is called the service provider, and the user of a service is called a service requestor. Services do not maintain any internal state. State information is either stored in a database or is maintained by the service requestor.

Microservices

Micro services are small-scale, stateless services that have a single responsibility. Software products that use microservices are said to have a microservices architecture. A microservices architecture is based on services that are fine-grain components with a single responsibility. If you need to create cloud-based software products, then it is recommend to use a microservice architecture.

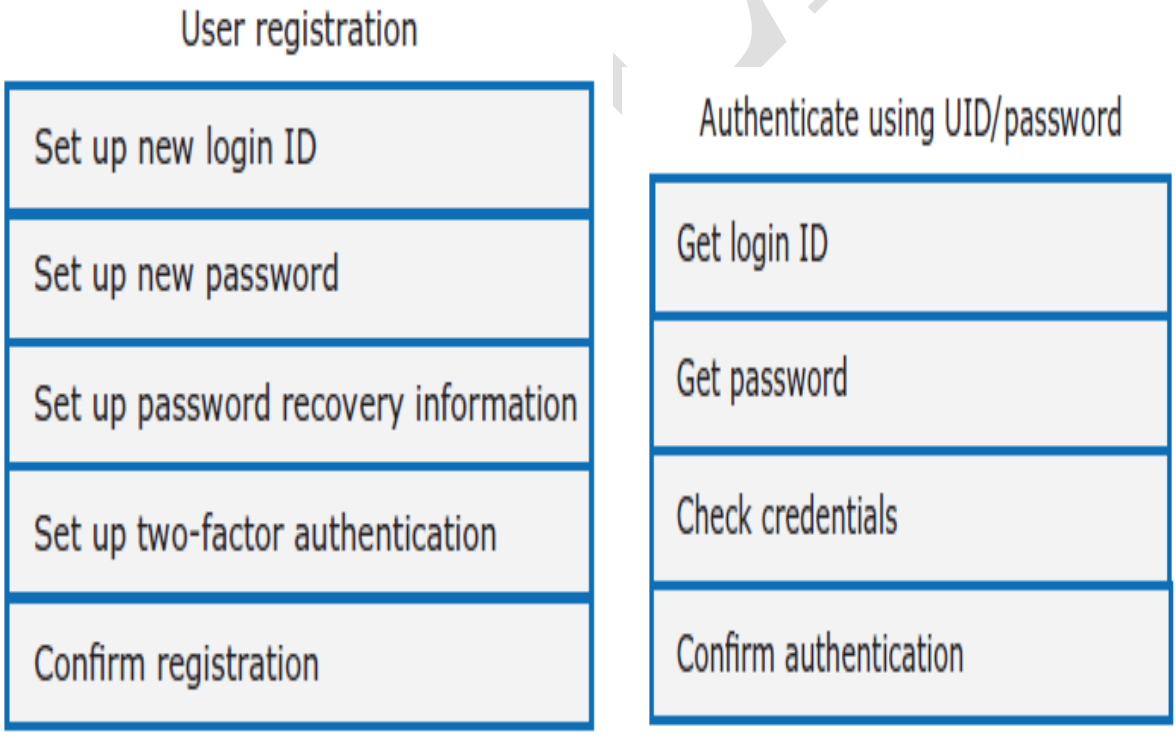
Example of microservices

Consider a system that uses an authentication module that provides the following features:

- User registration, where users provide information about their identity, Security information, mobile (cell) phone number, and email address;
- Authentication using user ID (UID)/password;
- Two-factor authentication using code sent to mobile phone;
- User information management—for example, ability to change password or mobile phone number.

In a microservices architecture, these features are too large to be microservices. To identify the microservices that might be used in the authentication system, you need to break down the coarse-grain features into more detailed functions. Below figure shows what these functions might be for user registration and UID/password authentication.

Fig: Functional breakdown of authentication features



Below figure shows the microservices that could be used to implement user authentication.

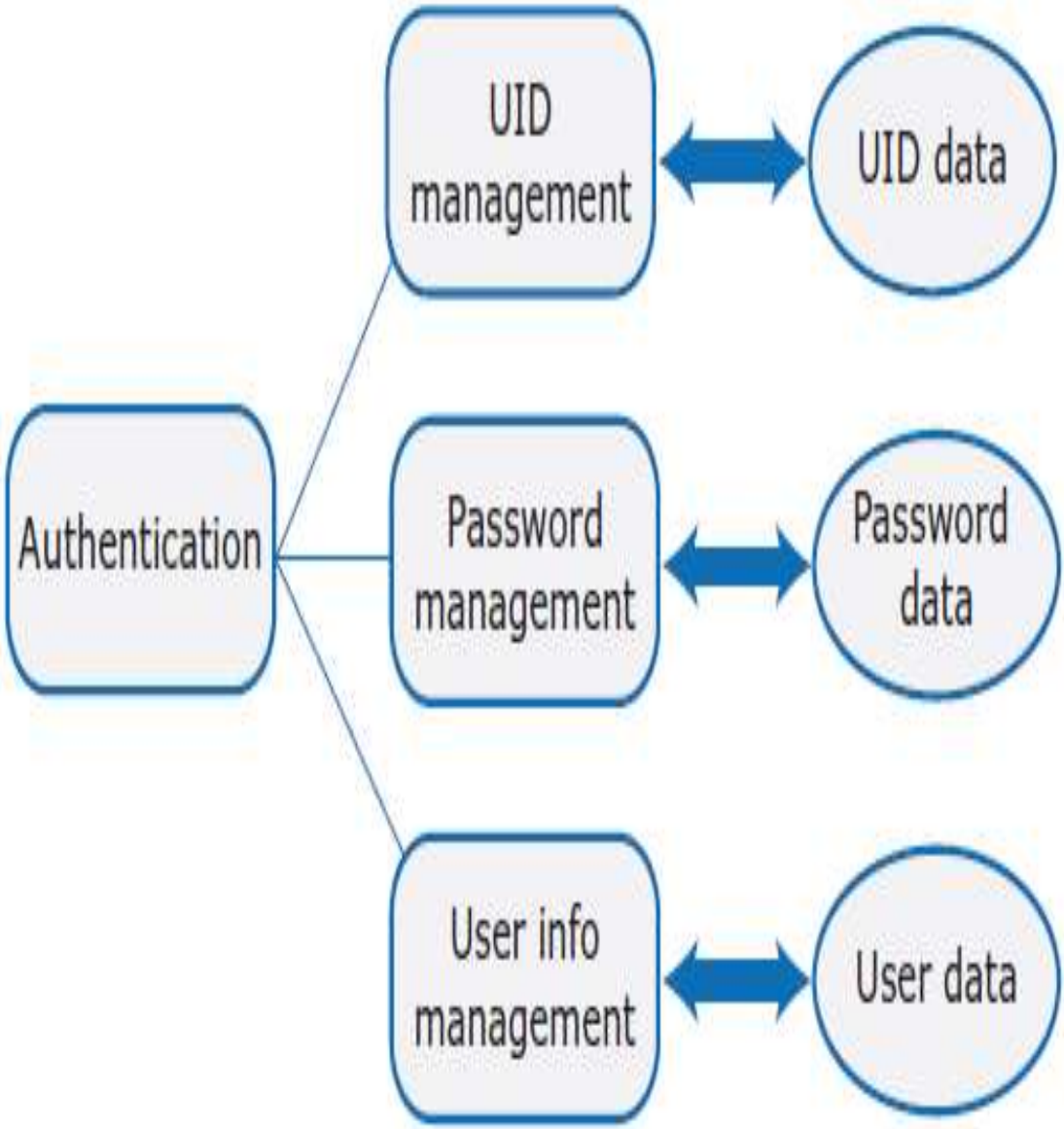


Figure: Authentication microservices

Characteristics of microservices are given below:

Characteristic	Explanation
Self-contained	Microservices do not have external dependencies. They manage their own data and implement their own user interface.
Lightweight	Microservices communicate using lightweight protocols, so that service communication overheads are low.
Implementation independent	Microservices may be implemented using different programming languages and may use different technologies (e.g., different types of database) in their implementation.
Independently deployable	Each microservice runs in its own process and is independently deployable, using automated systems.
Business-oriented	Microservices should implement business capabilities and needs, rather than simply provide a technical service.

- Microservices communicate by exchanging messages.
- A well-designed microservice should have high cohesion and low coupling.
- **Coupling** is a measure of the number of relationships that one component has with other components in the system. Low coupling means that components do not have many relationships with other components.
- **Cohesion** is a measure of the number of relationships that parts of a component have with each other. High cohesion means that all of the component parts that are needed to deliver the component‘s functionality are included in the component.
- Low coupling is important in microservices because it leads to independent services.
- High cohesion is important because it means that the service does not have to call lots of other services during execution. Calling other services involves communications overhead, which can slow down a system.

The aim of developing **highly cohesive services** has led to a fundamental principle that underlies microservice design: the **—single responsibility principle**.‖ Each element in a system should do one thing only. However, the problem with this is that —one thing only‖ is difficult to define in a way that is applicable to all services.

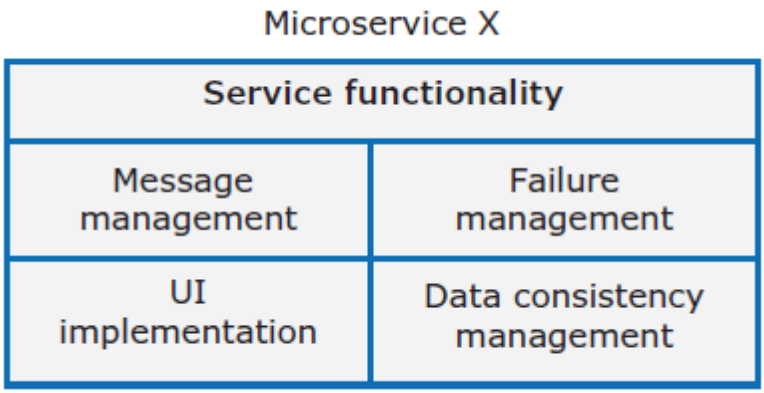
If you take the single responsibility principle literally, you would implement separate services for creating and changing a password and for checking that a password is correct. However, these simple services would all have to use a shared password database. This is undesirable because it increases the coupling between these services. Therefore responsibility should not always mean a single, functional activity. In this case, we can interpret a single responsibility as the responsibility to maintain stored passwords.

—Microservices‖ are small-scale components, so developers often ask **—How big should a microservice be**?‖

 It should be possible for a microservice to be developed, tested, and deployed by a service development team in two weeks or less.

 The team size should be such that the whole team can be fed by no more than two large pizzas (Amazon‘s guideline). This places an upper limit on the team size of eight to ten people (depending on how hungry they are).

The independence of microservices means that, each service has to include **support code**. These support codes are:



Message management code in a microservice is responsible for processing incoming and outgoing messages. Incoming messages have to be checked for validity. Outgoing messages have to be packed into the correct format for service communication.

Failure management code in a microservice has two concerns. First, it has to cope with circumstances where the microservice cannot properly complete a requested operation. Second, if external interactions are required, such as a call to another service, it has to handle the situation where that interaction does not succeed because the external service returns an error or does not reply.

Data consistency management is needed when the data used in a microservice are also used by other services. In those cases, there needs to be a way of communicating data updates between services and ensuring that the changes made in one service are reflected in all services that use the data.

UI implementation: For complete independence, each microservice should maintain its own user interface.

MICROSERVICES ARCHITECTURE

A microservices architecture is a tried and tested way of implementing a logical software architecture. This architectural style aims to address two fundamental problems with monolithic architecture:
1. When a monolithic architecture is used, the whole system has to be rebuilt, retested, and re-deployed when any change is made. This can be a slow process, as changes to one part of the system can adversely affect other components. Frequent application updates are therefore impossible.

2. As the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of system components that implement the most popular system functions.

Architectural design decisions

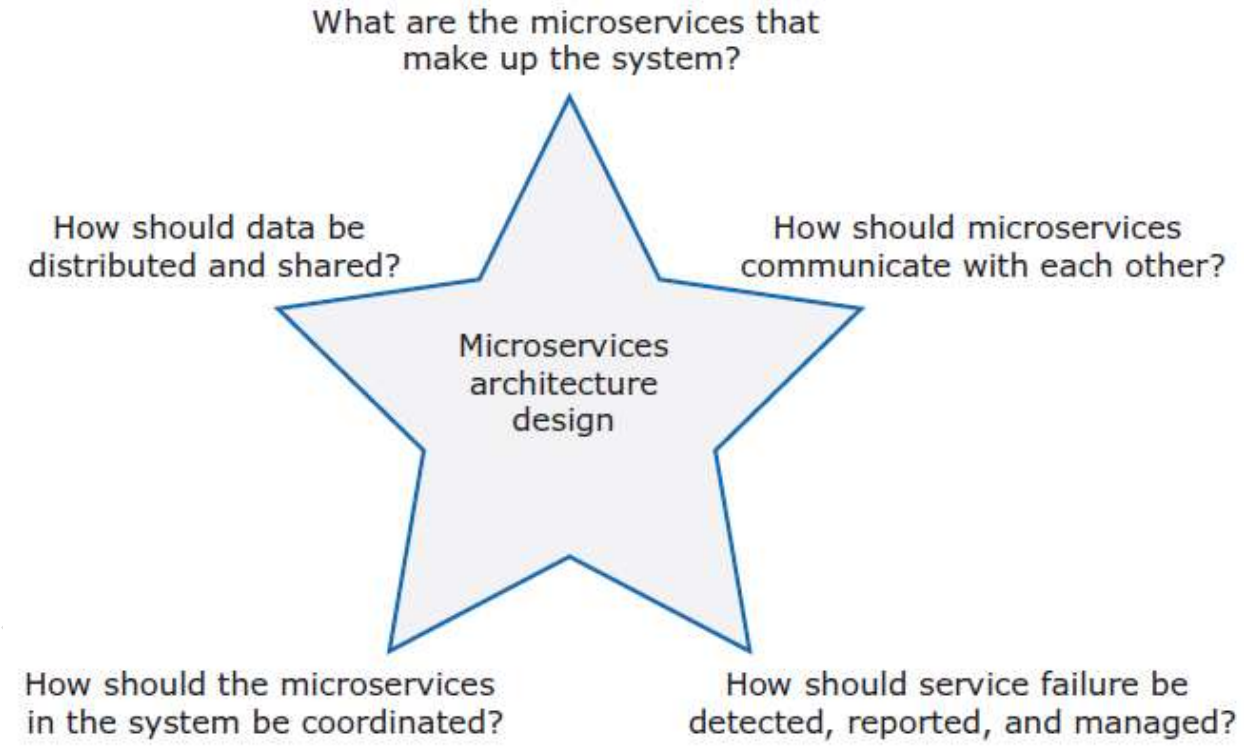


Fig: Key design issues for microservices architecture
One of the most important jobs for a system architect is to decide how the overall system should be decomposed into a set of microservices. For that there are some general guidelines:

1. ***Balance fine-grain functionality and system performance:*** If each of your services offers only a single, very specific service, however, it is inevitable that you will need to have more service communications to implement user functionality. This slows down a system.

2. ***Follow the "common closure principle":*** This means that elements of a system that are likely to be changed at the same time should be located within the same service.

3. ***Associate services with business capabilities****:* A business capability is a discrete area of business functionality that is the responsibility of an individual or a group. For example, the provider of a photo-printing system will have a group responsible for sending photos to users (dispatch capability), a set of printing machines (print capability), someone responsible for finance (payment service), and so on. You should identify the services that are required to support each business capability.

4. ***Design services so that they have access to only the data that they need***

Service communications:

Services communicate by exchanging messages. These messages include information about the originator of the message as well as the data that are the input to or output from the request.

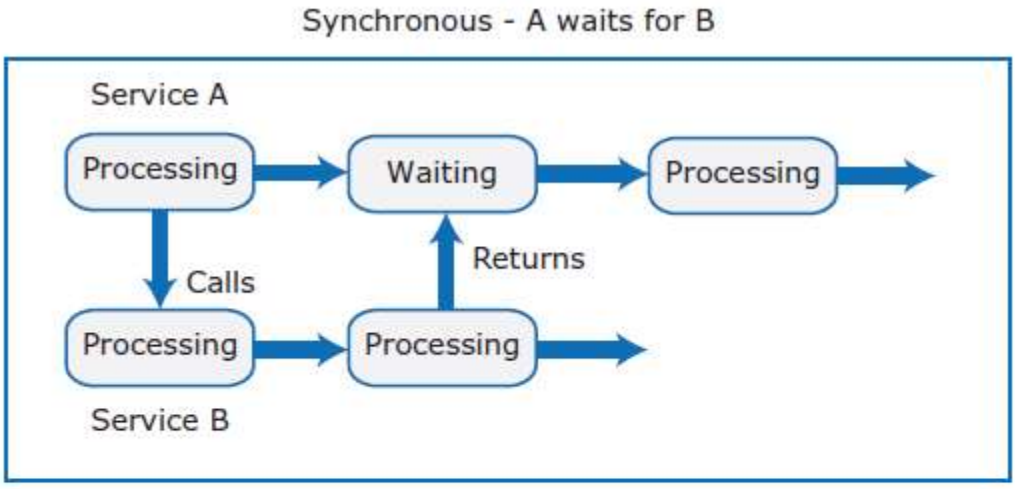
While establishing a standard for communication, the following key decisions must be taken:

- Should service interaction be synchronous or asynchronous?
- Should services communicate directly or via message broker middleware?
- What protocol should be used for messages exchanged between services?

In **a synchronous interaction**, service A issues a request to service B. Service A then suspends processing while service B is processing the request. It waits until service B has returned the required information before continuing execution.

In **an asynchronous interaction**, service A issues the request that is queued for processing by service B. Service A then continues processing without waiting for service B to finish its computations. Sometime later, service B completes the earlier request from service A and queues the result to be retrieved by service A. Service A therefore has to check its queue periodically to see if a result is available.

Synchronous interaction is less complex than asynchronous interaction. Consequently, synchronous programs are easier to write and understand. There will probably be fewer difficult-to-find bugs. On the other hand, asynchronous interaction is often more efficient than synchronous interaction, as services are not idle while waiting for a response.


Synchronous - A waits for B
Service A
Processing
Waiting
Processing
Calls
Returns
Processing
Processing
Service B

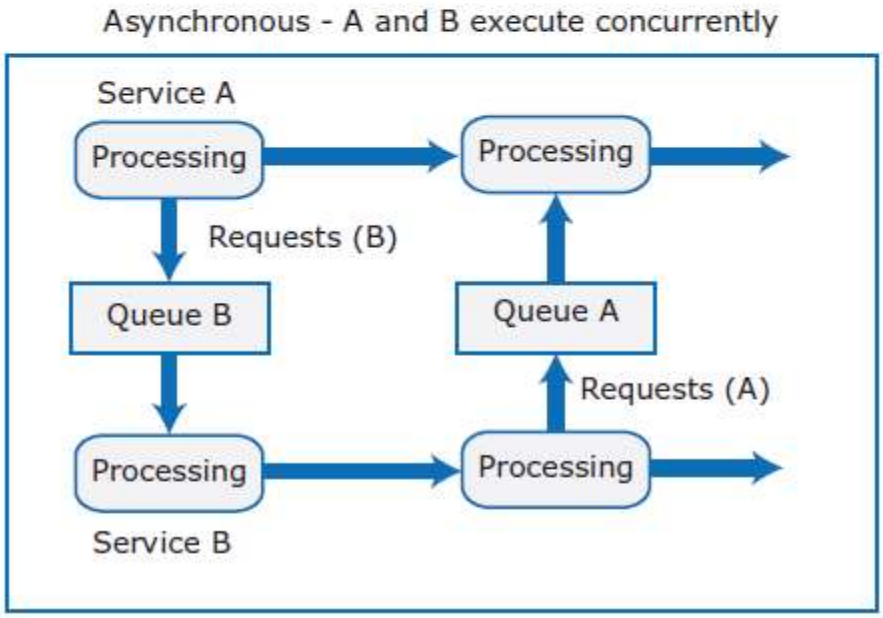

Asynchronous - A and B execute concurrently
Service A
Processing
Processing
Requests (B)
Queue B
Queue A
Requests (A)
Processing
Processing
Service B

Fig: Synchronous and asynchronous microservice interaction

Direct service communication requires that interacting services know each other‘s addresses. The services interact by sending requests directly to these addresses. **Indirect communication** involves naming the service that is required and sending that request to a message broker (sometimes called a message bus). The message broker is then responsible for finding the service that can fulfill the service request. Below figure shows these communication alternatives.

Direct service communication is usually faster, but it means that the requesting service must know the URI (uniform resource identifier) of the requested service. If, that URI changes, then the service request will fail. Indirect communication requires additional software (a message broker) but services are requested by name rather than a URI. The message broker finds the address of the requested service and directs the request to it

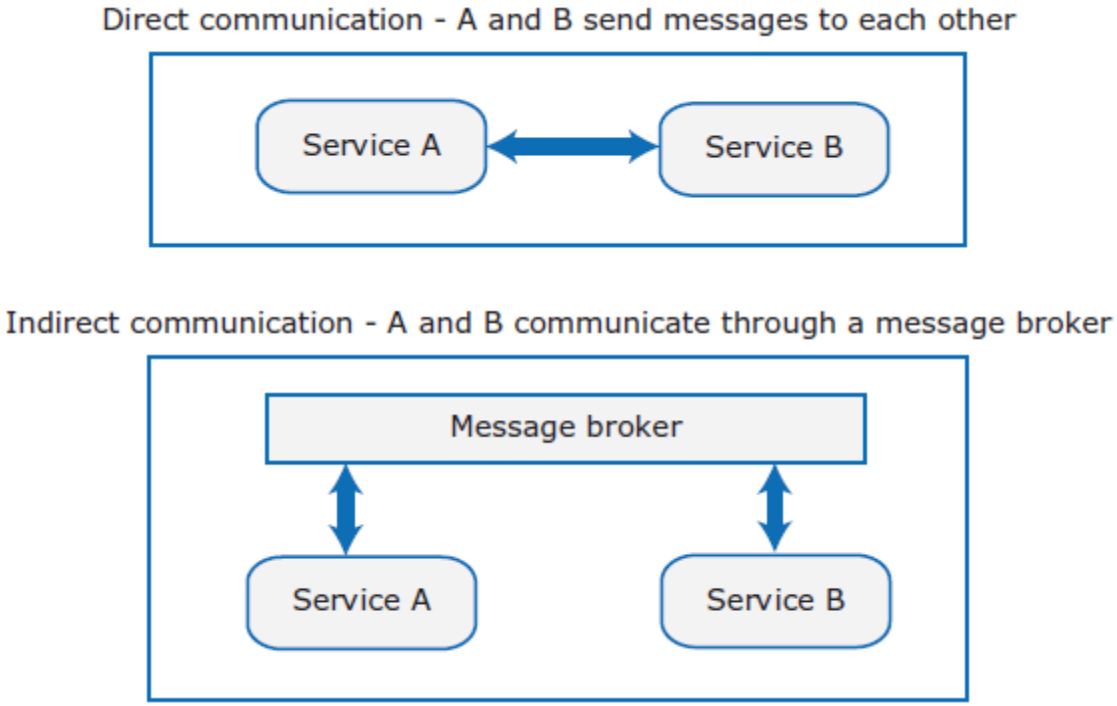


Fig: Direct and indirect service communication

Data distribution and sharing:

Each microservice should manage its own data. In an ideal world, the data managed by each service would be completely independent. There would be no need to propagate data changes made in one service to other services. However, in the real world, complete data independence is impossible.

You need to think about the microservices as an interacting system rather than as individual units. This means:

1. You should isolate data within each system service with as little data sharing as possible.
2. If data sharing is unavoidable, you should design microservices so that most sharing is read-only, with a minimal number of services responsible for data updates.
3. If services are replicated in your system, you must include a mechanism that can keep the database copies used by replica services consistent.

Access to the shared data is managed by a database management system (DBMS). Failure of services in the system and concurrent updates to shared data have the potential to cause database inconsistency.

Without controls, if services A and B are updating the same data, the value of that data depends on the timing of the updates. However, by using ACID properties, the DBMS serializes the updates and avoids inconsistency.

Systems that use microservices have to be designed to tolerate some degree of data inconsistency. The databases used by different services or service replicas need not be completely consistent all of the time.

Two types of inconsistency have to be managed:
1. ***Dependent data inconsistency*** The actions or failures of one service can cause the data managed by another service to become inconsistent.
2. ***Replica inconsistency*** Several replicas of the same service may be executing concurrently. These all have their own database copy and each updates its own copy of the service data. You need a way of making these databases ―eventually consistent‖ so that all replicas are working on the same data.

Service coordination:
Most user sessions involve a series of interactions in which operations have to be carried out in a specific order. This is called a workflow. As an example, the workflow for UID/password authentication in which there is a limited number of allowed authentication attempts is shown in below figure.

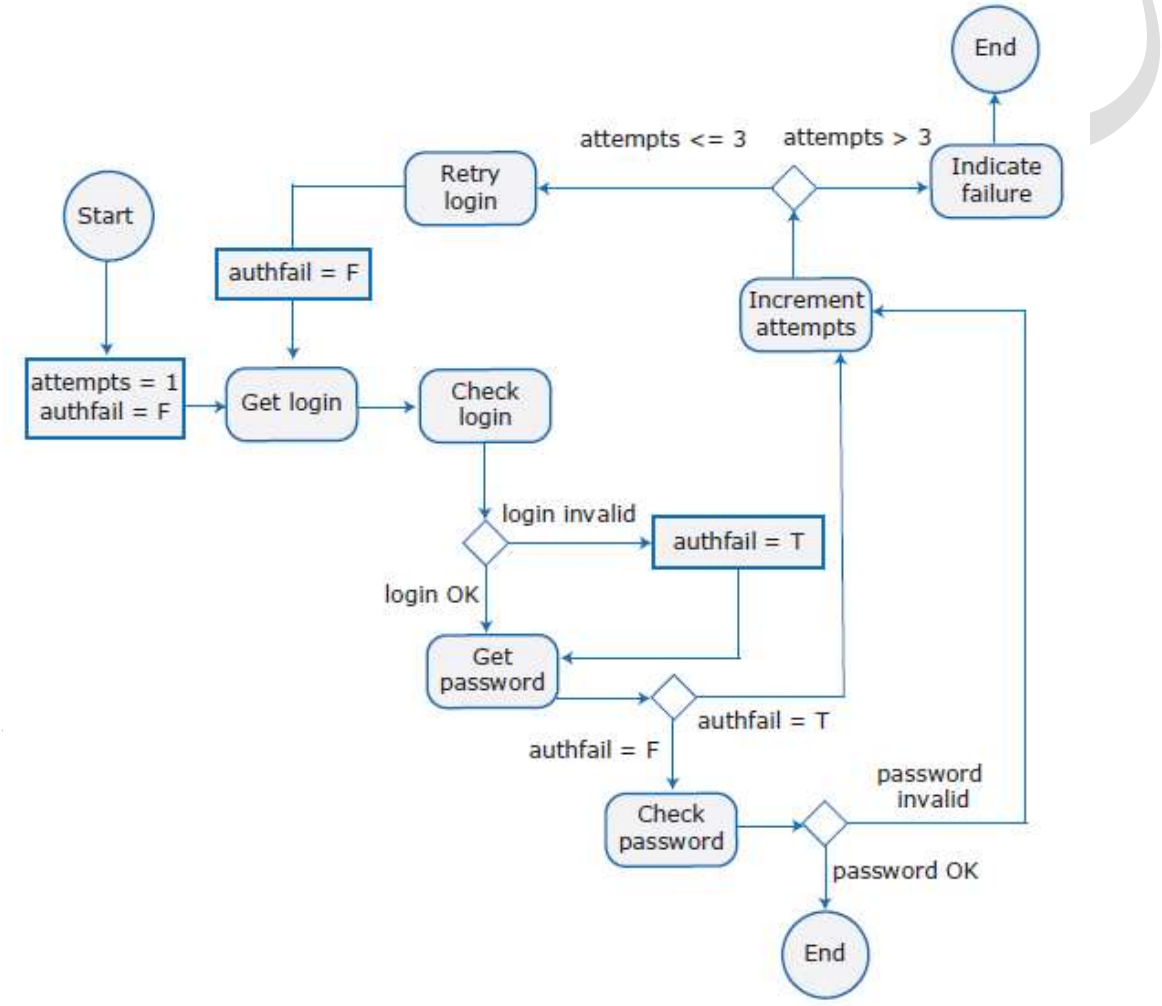


Fig: Authentication workflow

In this example workflow, the user is allowed three login attempts before the system indicates that the login has failed.

One way to **implement this workflow** is to define the workflow explicitly (either in a workflow language or in code) and to have a separate service that executes the workflow by calling the component services in turn. This is called ―**orchestration**,‖ reflecting the notion that an orchestra conductor instructs the musicians when to play their parts. In an orchestrated system, there is an overall controller.
An alternative approach is called **"choreography**.‖ This term is derived from dance rather than music, where there is no ―conductor‖ for the dancers. Rather, the dance proceeds as dancers observe one another. Their decision to move on to the next part of the dance depends on what the other dancers are doing.

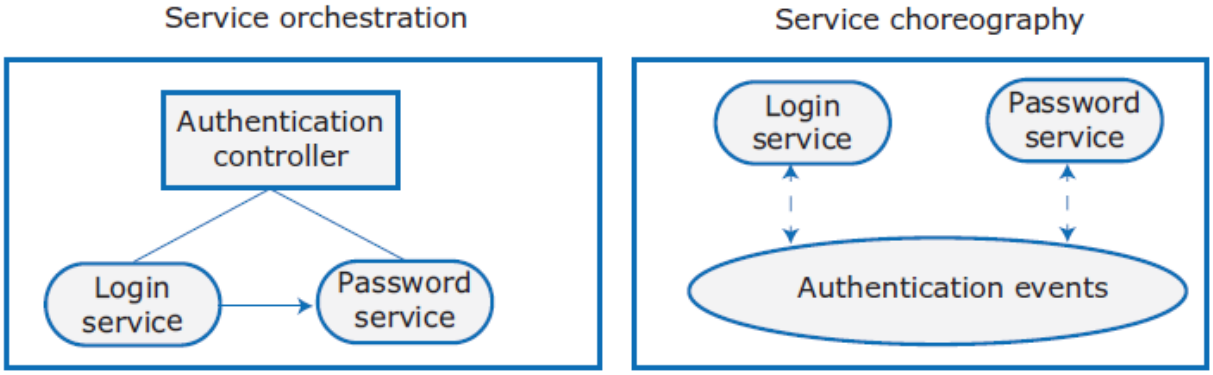


Fig: Orchestration and choreography

Choreography depends on each service emitting an event to indicate that it has completed its processing. Other services watch for events and react accordingly when events are observed.
There is no explicit service controller. To implement service choreography, you need additional software such as a message broker.

A problem with service choreography is that there is no simple correspondence between the workflow and the actual processing that takes place. This makes choreographed workflows harder to debug. If a failure occurs during workflow processing, it is not immediately obvious what service has failed.

Furthermore, recovering from a service failure is sometimes difficult to implement in a choreographed system.

In an orchestrated approach, if a service fails, the controller knows which service has failed and where the failure has occurred in the overall process.

Failure management:

The three kinds of failure with in a microservices system are shown in the below table.

Failure type	Explanation
Internal service failure	These are conditions that are detected by the service and can be reported to the service requestor in an error message. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.
External service failure	These failures have an external cause that affects the availability of a service. Failure may cause the service to become unresponsive and actions have to be taken to restart the service.
Service performance failure	The performance of the service degrades to an unacceptable level. This may be due to a heavy load or an internal problem with the service. External service monitoring can be used to detect performance failures and unresponsive services.

The simplest way to report microservice failures is to use HTTP status codes, which indicate whether or not a request has succeeded.
One way to discover whether a service that you are requesting is unavailable or running slowly is to put a timeout on the request. A timeout is a counter that is associated with the service requests and starts running when the request is made. Once the counter reaches some predefined value, such as 10 seconds, the calling service assumes that the service request has failed and acts accordingly.

SERVICE DEPLOYMENT (MICROSERVICE DEPLOYEMENT)

After a system has been developed and delivered, it has to be deployed on servers, monitored for problems, and updated as new versions become available. The development team is responsible for deployment and service management as well as software development. That is, in microservice architecture, we use the concepts of DevOps - a combination of Development and Operations, for software deployment.
In this area, a good practise is to adopt a policy of continuous deployment. Continuous deployment means that as soon as a change to a service has been made and validated, the modified service is re-deployed.
Continuous deployment depends on automation so that as soon as a change is committed, a series of automated activities is triggered to test the software. If the software ―passes‖ these tests, it then enters another automation pipeline that packages and deploys the software.

Containers are usually the best way to package a cloud service for deployment. Below figure is a simplified diagram of the continuous deployment process.

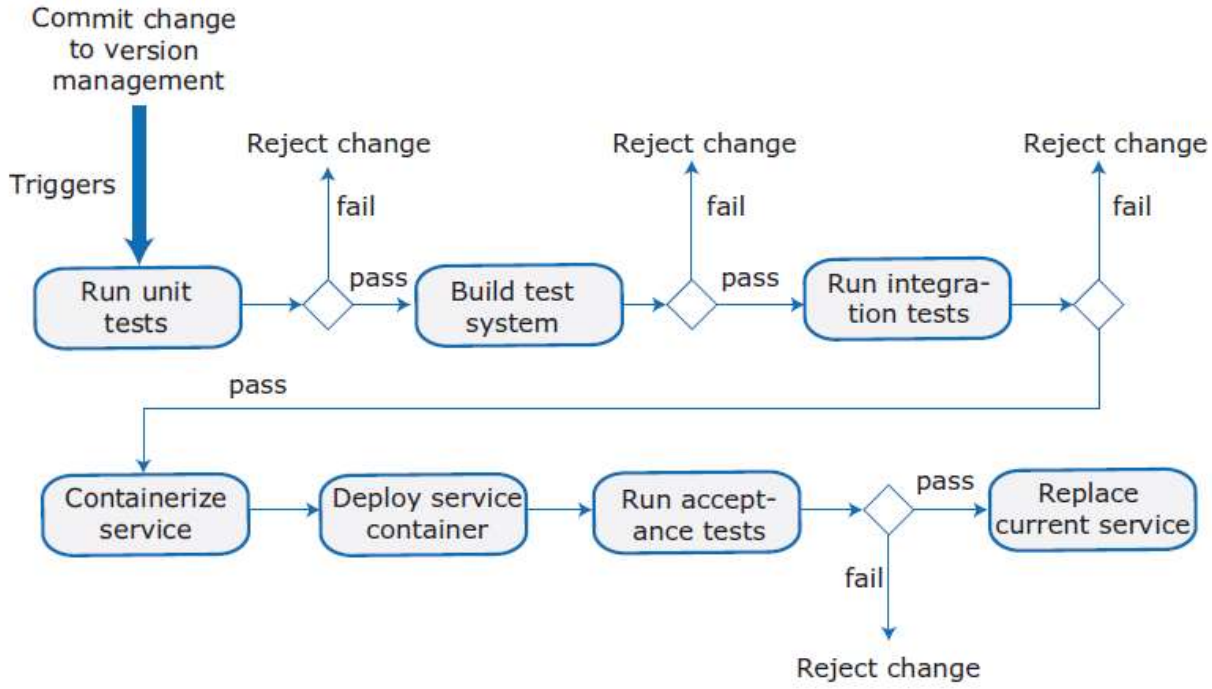


Fig: A continuous deployment pipeline

CONTENT BEYOND SYLLABUS

1. Cloud Hypervisor.
2. Edge computing.

What is a Cloud Hypervisor?

A **Cloud Hypervisor** is software that enables the sharing of cloud provider's physical compute and memory resources across multiple virtual machines (VMs). Originally created for mainframe computers in the 1960s, hypervisors gained wide popularity with the introduction of VMware for industry standard servers in the 1990s, enabling a single physical server to independently run multiple guest VMs each with their own operating systems (OSs) that are logically separate from each other. In this manner, problems or crashes in one guest VM have no effect on the other guest VMs, OSs, or the applications running on them.

Although there are multiple types of VMs, they all perform the same task, enabling a single set of physical server hardware (including CPU, memory, storage, and peripherals) and enabling the simultaneous use by multiple instances of OSs, whether Windows, Linux, or both.

Why is a Cloud Hypervisor important?

Just as hypervisors make it possible to gain a new level of computer utilization, a Cloud Hypervisor is the underpinning of all cloud compute offerings, enabling VMs and containers to run side-by-side on a single server, whether those VMs belong to a single client or to multiple clients of the cloud provider. It is this multitenancy that powers the economics for most cloud compute offerings.

Hypervisors and the VMs they support provide the portability that enables workloads to easily be migrated between cloud providers and on-premises servers. This enables organizations to rapidly scale from on-premises servers to cloud providers or to add more instances of applications already running in the cloud when spikes in demand occur.

Cloud Hypervisors help cloud providers reduce the amount of space servers use, while reducing the amount of energy needed to power and cool the vast array of servers under their management.

How does a Cloud Hypervisor work?

Cloud Hypervisors abstract the underlying servers from 'Guest' VMs and OSs. OS calls for server resources (CPU, memory, disk, print, etc) are intercepted by the Cloud Hypervisor which allocates resources and prevents conflicts. As a rule, guest VMs and OSs run in a less-privileged mode than the hypervisor so they cannot impact the operation of the hypervisor or other guest VMs

There are two major classifications of Hypervisor: Bare metal or native (Type 1) and Hosted (Type 2). Type 1 Hypervisors run directly on host machine hardware with no OS beneath. These hypervisors communicate directly with the host machine resources. VMware ESXi and Microsoft Hyper-V are Type 1.

Type 2 Hypervisors usually run above the host machine OS and rely on the host OS for access to machine resources. They are easier to se up and manage since the OS is already in place, and thus Type 2 hypervisors are often used for home use and for testing VM functionality. VMware Player and VMware Workstation are Type 2 hypervisors.

KVM (Kernel-based Virtual Machine) is a popular hybrid hypervisor with some Type 1 and Type 2 characteristics. This open-source hypervisor it built into Linux and lets Linux act as a Type 1 hypervisor and an OS at the same time.

What are the Benefits of a Cloud Hypervisor?

There are several benefits to using a hypervisor that hosts multiple virtual machines:

Time to Use: Cloud Hypervisors enable VMs to be instantly spun up or down, as opposed to days or weeks required to deploy a bare metal server. This enables projects to be created and have teams working the same day. Once a project is complete, VMs can be terminated to save organizations from paying for unnecessary infrastructure.

Utilization: Cloud Hypervisors enable several VMs to run on a single physical server and for all the VMs to share its resources. This improves the server utilization and saves on power, cooling, and real estate that is no longer needed for each individual VM.

Flexibility: Most Cloud Hypervisors are Type 1 (Bare-metal) enabling guest VMs and OSs to execute on a broad variety of hardware, since the hypervisor abstracts the VMs from the underlying machine's drivers and devices.

Portability: Since Cloud Hypervisors enable portability of workloads between VMs or between a VM and an organization's on-premises hardware. Applications that are seeing spikes in demand can simply access additional machines to scale as needed.

Reliability: Hardware failures can be remediated by moving VMs to other machines, either at the cloud provider or in a private cloud or on-premises hardware. Once the failure is repaired workloads can fail back to ensure availability of application resources on the VM.

What are Types of Hypervisors in Cloud Computing?

There are two main hypervisor types, referred to as "Type 1" "bare metal") and "Type 2" (or "hosted"). A type 1 hypervisor acts like a lightweight operating system and runs directly on the host's hardware, while a type 2 hypervisor runs as a software layer on an operating system, like other computer programs.

Cloud providers most commonly deploy a Type 1 or bare-metal hypervisor, where virtualization software is installed directly on the hardware where the operating system is normally installed. Because bare-metal hypervisors are isolated from the attack-prone operating system, they are extremely secure. In addition, they generally perform better and more efficiently than hosted hypervisors. For these reasons, most enterprise companies choose bare-metal hypervisors for data center computing needs.

While bare-metal hypervisors run directly on the computing hardware, hosted or Type 2 hypervisors run on top of the operating system (OS) of the host machine. Although hosted hypervisors run within the OS, additional (and different) operating systems can be installed on top of the hypervisor. The downside of hosted hypervisors is that latency is higher than bare-metal hypervisors. This is because communication between the hardware and the hypervisor must pass through the extra layer of the OS. Hosted hypervisors are sometimes known as client hypervisors because they are most often used with end users and software testing, where higher latency is less of a concern.

Both types of hypervisors can run multiple virtual servers for multiple tenants on one physical machine. Public cloud service providers lease server space on the different virtual servers to different companies. One server might host several virtual servers that are all running workloads for different companies. This type of resource sharing can result in a “noisy neighbor” effect, when one of the tenants runs a large workload that interferes with the server performance for other tenants. It also poses more of a security risk than using a dedicated bare-metal server.

A Cloud Hypervisor comparison of major cloud providers demonstrates their similarity.

Amazon AWS EC2 uses a Cloud Hypervisor that is a customized version of the Xen hypervisor that takes advantage of paravirtualization for Linux guest VMs.

The Google Cloud Platform (GCP) Cloud Hypervisor is also based on the open-source KVM hypervisor; Google also invests in additional security hardening and protection and contributes their changes back to the KVM project for the benefit of all.

The Microsoft Azure Cloud Hypervisor is based on Microsoft Hyper-V, another Type 1 hypervisor popular in Windows environments and customized for the Microsoft Azure platform.

edge computing

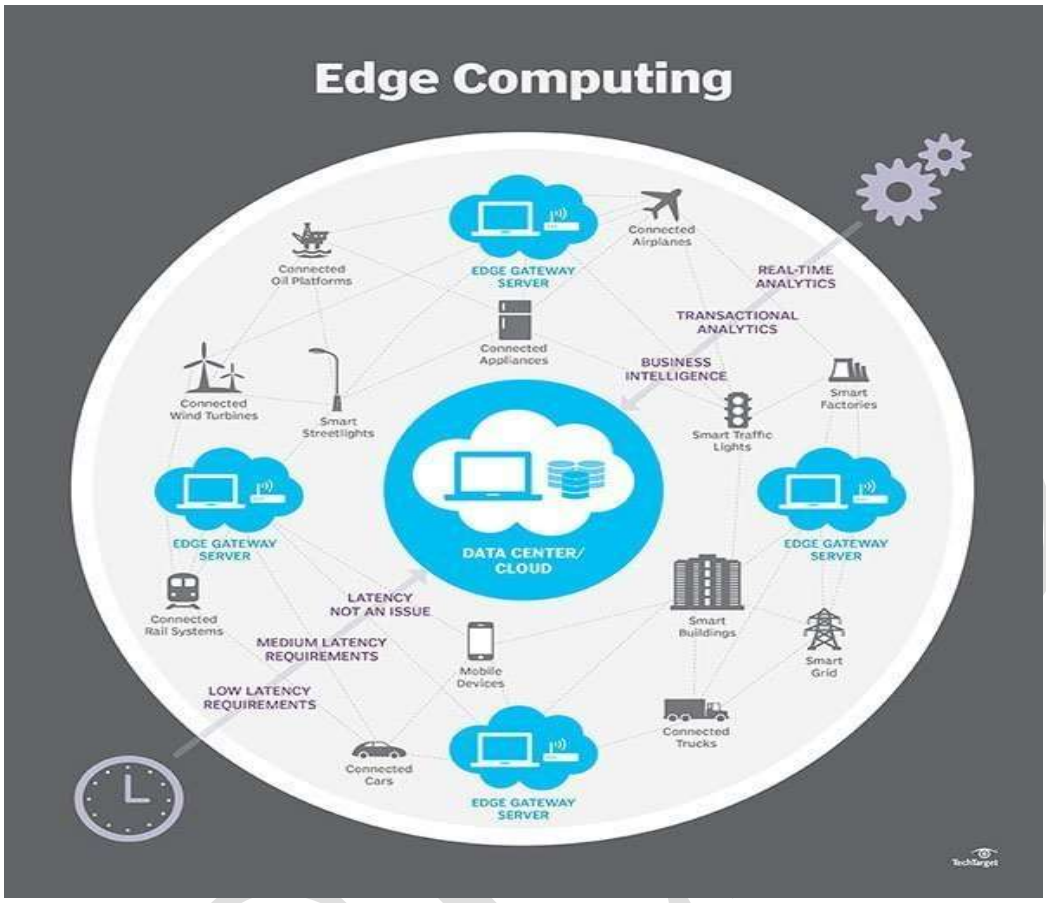
Edge computing is a distributed information technology (IT) architecture in which client data is processed at the periphery of the network, as close to the originating source as possible.

Data is the lifeblood of modern business, providing valuable business insight and supporting real-time control over critical business processes and operations. Today's businesses are awash in an ocean of data, and huge amounts of data can be routinely collected from sensors and IoT devices operating in real time from remote locations and inhospitable operating environments almost anywhere in the world.

But this virtual flood of data is also changing the way businesses handle computing. The traditional computing paradigm built on a centralized data center and everyday internet isn't well suited to moving endlessly growing rivers of real-world data. Bandwidth limitations, latency issues and unpredictable network disruptions can all conspire to impair such efforts. Businesses are responding to these data challenges through the use of edge computing architecture.

In simplest terms, edge computing moves some portion of storage and compute resources out of the central data center and closer to the source of the data itself. Rather than transmitting raw data to a central data center for processing and analysis, that work is instead performed where the data is actually generated -- whether that's a retail store, a factory floor, a sprawling utility or across a smart city. Only the result of that computing work at the edge, such as real-time business insights, equipment maintenance predictions or other actionable answers, is sent back to the main data center for review and other human interactions.

Thus, edge computing is reshaping IT and business computing. Take a comprehensive look at what edge computing is, how it works, the influence of the cloud, edge use cases, tradeoffs and implementation considerations.



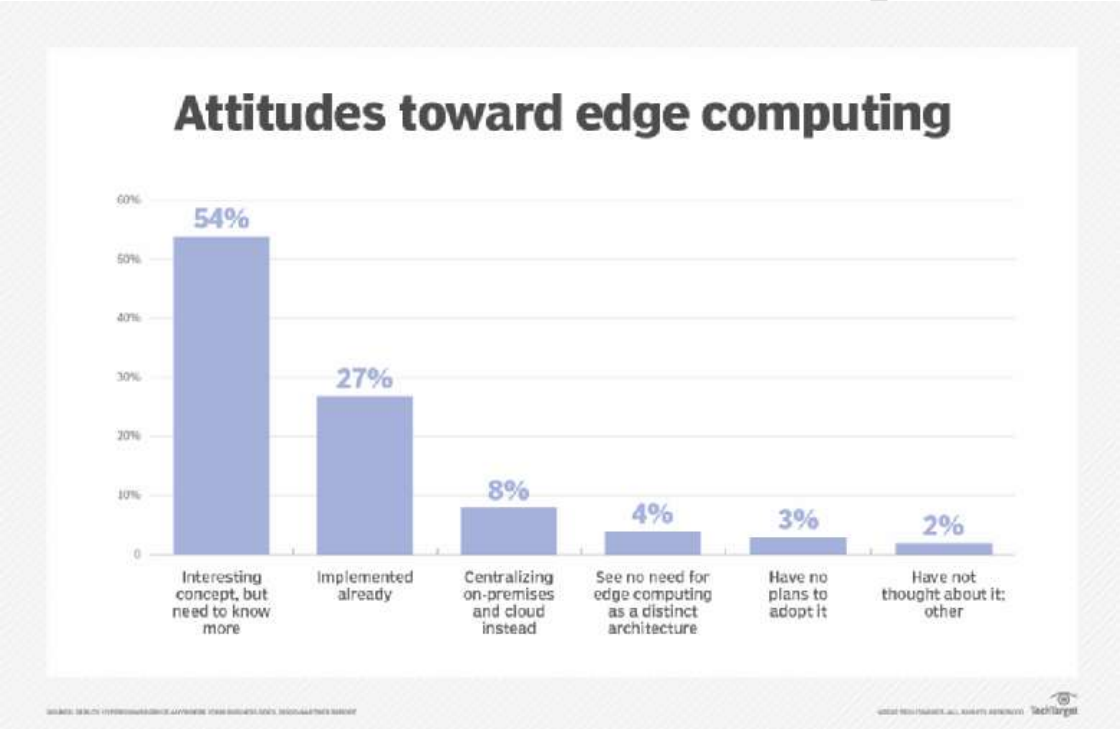
Edge computing brings data processing closer to the data source.

How does edge computing work?

Edge computing is all a matter of location. In traditional enterprise computing, data is produced at a client endpoint, such as a user's computer. That data is moved across a WAN such as the internet, through the corporate LAN, where the data is stored and worked upon by an enterprise application. Results of that work are then conveyed back to the client endpoint. This remains a proven and time-tested approach to client-server computing for most typical business applications.

But the number of devices connected to the internet, and the volume of data being produced by those devices and used by businesses, is growing far too quickly for traditional data center infrastructures to accommodate. Gartner predicted that by 2025, 75% of enterprise-generated data will be created outside of centralized data centers. The prospect of moving so much data in situations that can often be time- or disruption-sensitive puts incredible strain on the global internet, which itself is often subject to congestion and disruption.

So IT architects have shifted focus from the central data center to the logical *edge* of the infrastructure -- taking storage and computing resources from the data center and moving those resources to the point where the data is generated. The principle is straightforward: If you can't get the data closer to the data center, get the data center closer to the data. The concept of edge computing isn't new, and it is rooted in decades-old ideas of remote computing -- such as remote offices and branch offices -- where it was more reliable and efficient to place computing resources at the desired location rather than rely on a single central location.



Although only 27% of respondents have already implemented edge computing technologies, 54% find the idea interesting.

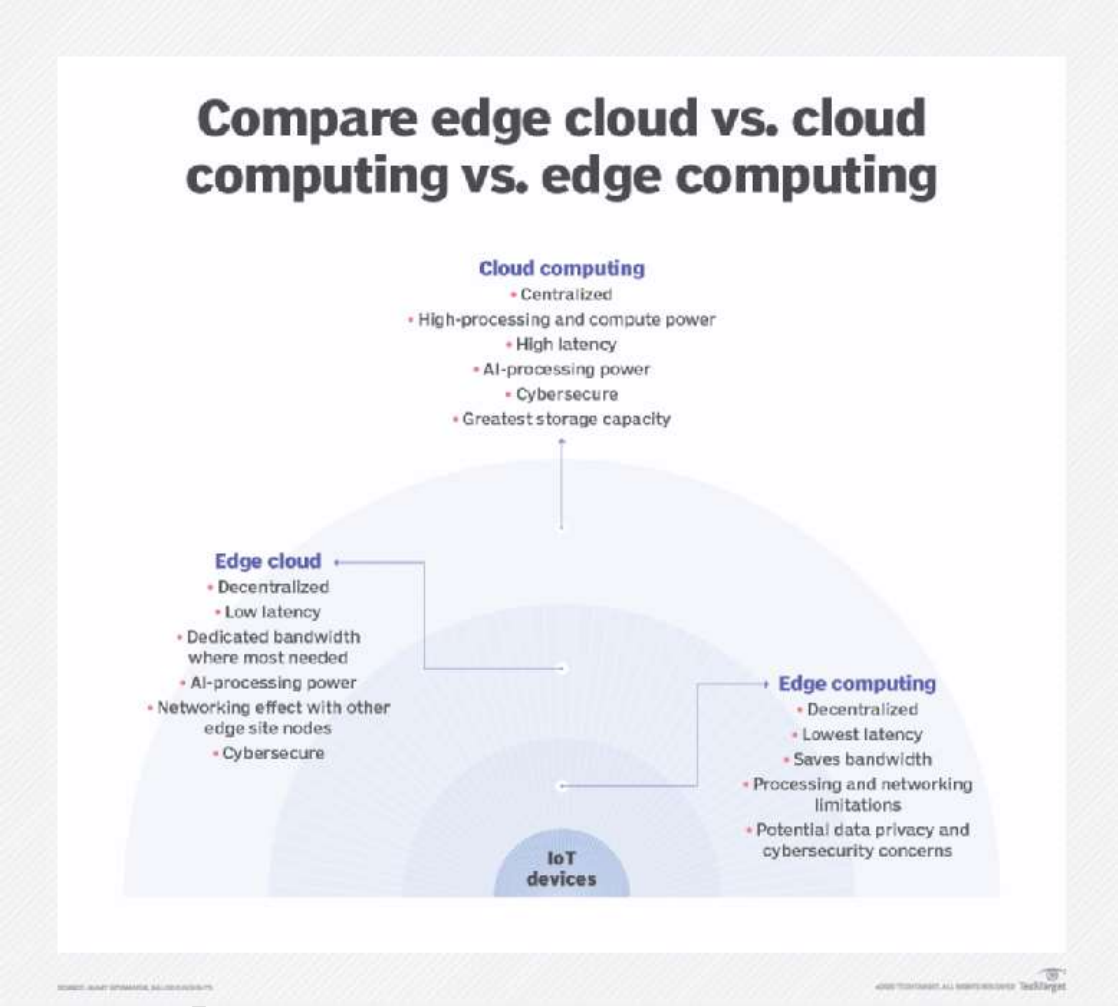
Edge computing puts storage and servers where the data is, often requiring little more than a partial rack of gear to operate on the remote LAN to collect and process the data locally. In many cases, the computing gear is deployed in shielded or hardened enclosures to protect the gear from extremes of temperature, moisture and other environmental conditions. Processing often involves normalizing and analyzing the data stream to look for business intelligence, and only the results of the analysis are sent back to the principal data center.

The idea of business intelligence can vary dramatically. Some examples include retail environments where video surveillance of the showroom floor might be combined with actual sales data to determine the most desirable product configuration or consumer demand. Other examples involve predictive analytics that can guide equipment maintenance and repair before actual defects or failures occur. Still other examples are often aligned with utilities, such as water treatment or electricity generation, to ensure that equipment is functioning properly and to maintain the quality of output.

Edge vs. cloud vs. fog computing

Edge computing is closely associated with the concepts of *cloud computing* and *fog computing*. Although there is some overlap between these concepts, they aren't the same thing, and generally shouldn't be used interchangeably. It's helpful to compare the concepts and understand their differences.

One of the easiest ways to understand the differences between edge, cloud and fog computing is to highlight their common theme: All three concepts relate to distributed computing and focus on the physical deployment of compute and storage resources in relation to the data that is being produced. The difference is a matter of where those resources are located.



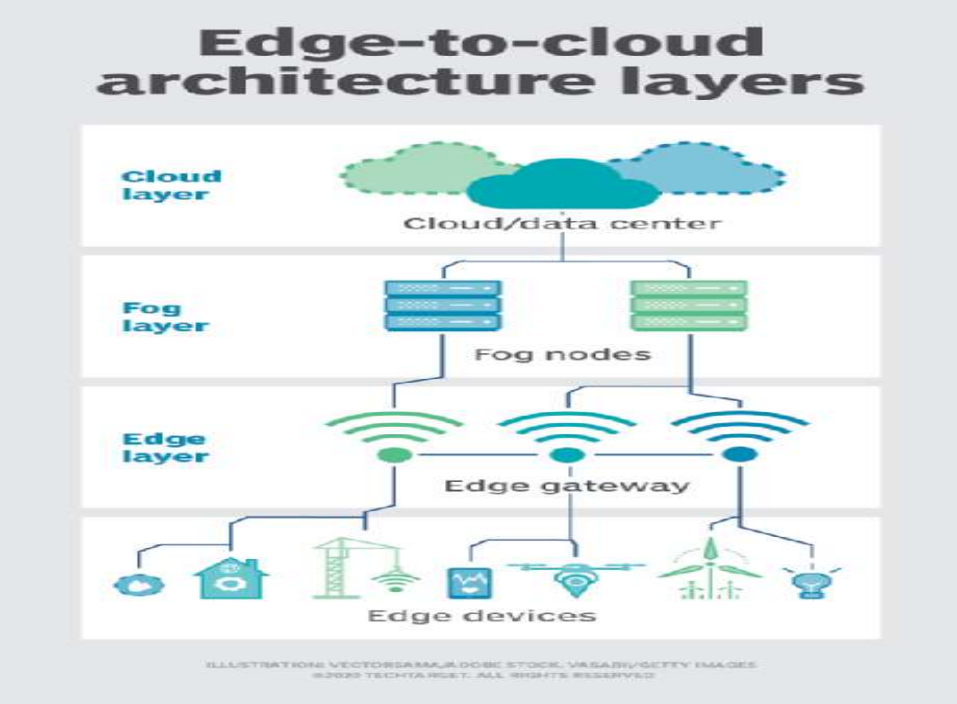
Compare edge cloud, cloud computing and edge computing to determine which model is best for you.

Edge.

Edge computing is the deployment of computing and storage resources at the location where data is produced. This ideally puts compute and storage at the same point as the data source at the network edge. For example, a small enclosure with several servers and some storage might be installed atop a wind turbine to collect and process data produced by sensors within the turbine itself. As another example, a railway station might place a modest amount of compute and storage within the station to collect and process myriad track and rail traffic sensor data. The results of any such processing can then be sent back to another data center for human review, archiving and to be merged with other data results for broader analytics.

Cloud.

Cloud computing is a huge, highly scalable deployment of compute and storage resources at one of several distributed global locations (regions). Cloud providers also incorporate an assortment of pre-packaged services for IoT operations, making the cloud a preferred centralized platform for IoT deployments. But even though cloud computing offers far more than enough resources and services to tackle complex analytics, the closest regional cloud facility can still be hundreds of miles from the point where data is collected, and connections rely on the same temperamental internet connectivity that supports traditional data centers. In practice, cloud computing is an alternative -- or sometimes a complement -- to traditional data centers. The cloud can get centralized computing much closer to a data source, but not at the network edge.



Unlike cloud computing, edge computing allows data to exist closer to the data sources through a network of edge devices.

Fog. But the choice of compute and storage deployment isn't limited to the cloud or the edge. A cloud data center might be too far away, but the edge deployment might simply be too resource-limited, or physically scattered or distributed, to make strict edge computing practical. In this case, the notion of fog computing can help. Fog computing typically takes a step back and puts compute and storage resources "within" the data, but not necessarily "at" the data.

Fog computing environments can produce bewildering amounts of sensor or IoT data generated across expansive physical areas that are just too large to define an *edge*. Examples include smart buildings, smart cities or even smart utility grids. Consider a smart city where data can be used to track, analyze and optimize the public transit system, municipal utilities, city services and guide long-term urban planning. A single edge deployment simply isn't enough to handle such a load, so fog computing can operate a series of fog node deployments within the scope of the environment to collect, process and analyze data.

Note: It's important to repeat that fog computing and edge computing share an almost identical definition and architecture, and the terms are sometimes used interchangeably even among technology experts.

Why is edge computing important?

Computing tasks demand suitable architectures, and the architecture that suits one type of computing task doesn't necessarily fit all types of computing tasks. Edge computing has emerged as a viable and important architecture that supports distributed computing to deploy compute and storage resources closer to -- ideally in the same physical location as -- the data source. In general, distributed computing models are hardly new, and the concepts of remote offices, branch offices, data center colocation and cloud computing have a long and proven track record.

But decentralization can be challenging, demanding high levels of monitoring and control that are easily overlooked when moving away from a traditional centralized computing model. Edge computing has become relevant because it offers an effective solution to emerging network problems associated with moving enormous volumes of data that today's organizations produce and consume. It's not just a problem of amount. It's also a matter of time; applications depend on processing and responses that are increasingly time-sensitive.

Consider the rise of self-driving cars. They will depend on intelligent traffic control signals. Cars and traffic controls will need to produce, analyze and exchange data in real time. Multiply this requirement by huge numbers of autonomous vehicles, and the scope of the potential problems becomes clearer. This demands a fast and responsive network. Edge -- and fog-- computing addresses three principal network limitations: bandwidth, latency and congestion or reliability.

- **Bandwidth.** Bandwidth is the amount of data which a network can carry over time, usually expressed in bits per second. All networks have a limited bandwidth, and the limits are more severe for wireless communication. This means that there is a finite limit to the amount of data -- or the number of devices -- that can communicate data across the network. Although it's possible to increase network bandwidth to accommodate more devices and data, the cost can be significant, there are still (higher) finite limits and it doesn't solve other problems.

- **Latency.** Latency is the time needed to send data between two points on a network. Although communication ideally takes place at the speed of light, large physical distances coupled with network congestion or outages can delay data movement across the network. This delays any analytics and decision-making processes, and reduces the ability for a system to respond in real time. It even cost lives in the autonomous vehicle example.
- **Congestion.** The internet is basically a global "network of networks." Although it has evolved to offer good general-purpose data exchanges for most everyday computing tasks -- such as file exchanges or basic streaming -- the volume of data involved with tens of billions of devices can overwhelm the internet, causing high levels of congestion and forcing time-consuming data retransmissions. In other cases, network outages can exacerbate congestion and even sever communication to some internet users entirely - making the internet of things useless during outages.

By deploying servers and storage where the data is generated, edge computing can operate many devices over a much smaller and more efficient LAN where ample bandwidth is used exclusively by local data-generating devices, making latency and congestion virtually nonexistent. Local storage collects and protects the raw data, while local servers can perform essential edge analytics -- or at least pre-process and reduce the data -- to make decisions in real time before sending results, or just essential data, to the cloud or central data center.

Edge computing use cases and examples

In principal, edge computing techniques are used to collect, filter, process and analyze data "in-place" at or near the network edge. It's a powerful means of using data that can't be first moved to a centralized location -- usually because the sheer volume of data makes such moves cost-prohibitive, technologically impractical or might otherwise violate compliance obligations, such as data sovereignty. This definition has spawned myriad real-world examples and use cases:

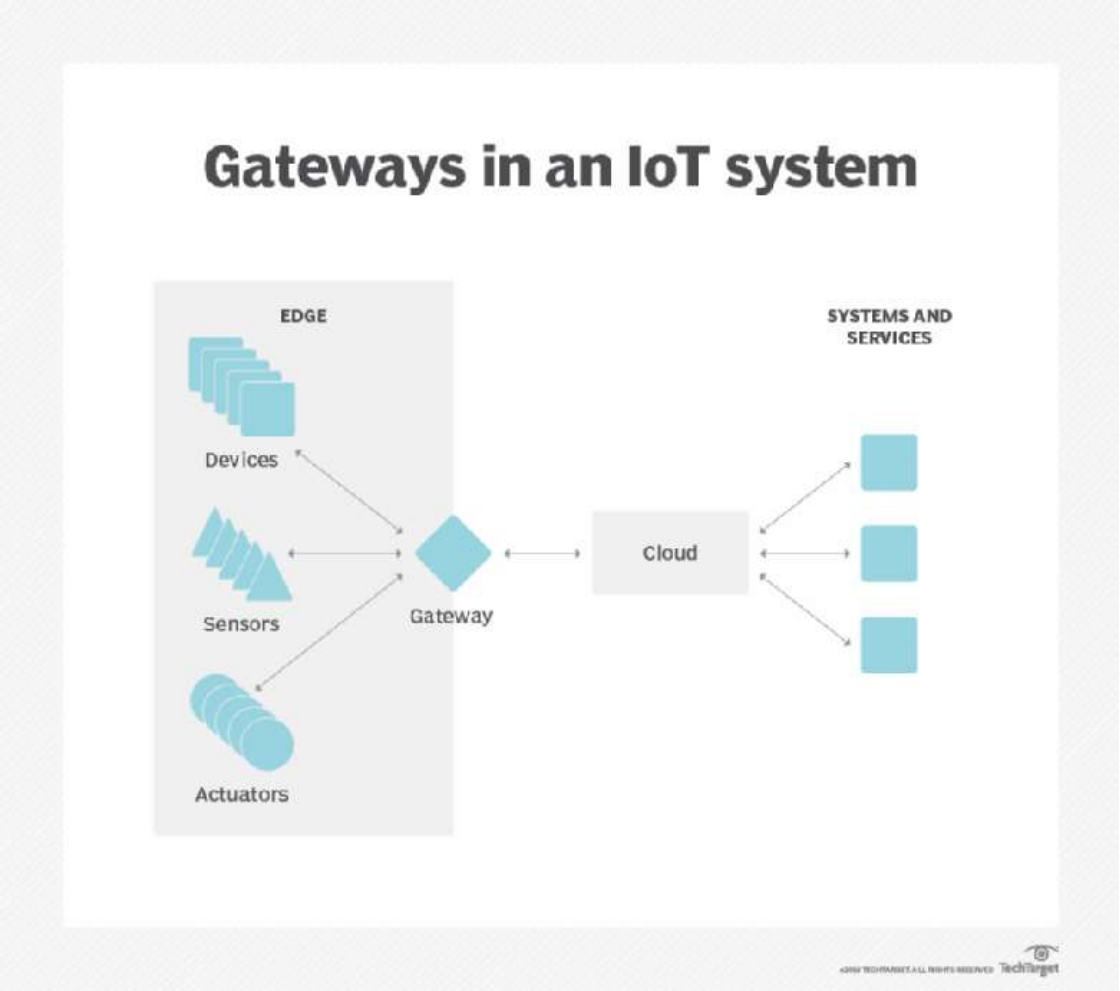
1. **Manufacturing.** An industrial manufacturer deployed edge computing to monitor manufacturing, enabling real-time analytics and machine learning at the edge to find production errors and improve product manufacturing quality. Edge computing supported the addition of environmental sensors throughout the manufacturing plant, providing insight into how each product component is assembled and stored -- and how long the components remain in stock. The manufacturer can now make faster and more accurate business decisions regarding the factory facility and manufacturing operations.
2. **Farming.** Consider a business that grows crops indoors without sunlight, soil or pesticides. The process reduces grow times by more than 60%. Using sensors enables the business to track water use, nutrient density and determine optimal harvest. Data is collected and analyzed to find the effects of environmental factors and continually improve the crop growing algorithms and ensure that crops are harvested in peak condition.
3. **Network optimization.** Edge computing can help optimize network performance by measuring performance for users across the internet and then employing analytics to determine the most reliable, low-latency network path for each user's traffic. In effect, edge computing is used to "steer" traffic across the network for optimal time-sensitive traffic performance.

4. **Workplace safety.** Edge computing can combine and analyze data from on-site cameras, employee safety devices and various other sensors to help businesses oversee workplace conditions or ensure that employees follow established safety protocols -- especially when the workplace is remote or unusually dangerous, such as construction sites or oil rigs.
5. **Improved healthcare.** The healthcare industry has dramatically expanded the amount of patient data collected from devices, sensors and other medical equipment. That enormous data volume requires edge computing to apply automation and machine learning to access the data, ignore "normal" data and identify problem data so that clinicians can take immediate action to help patients avoid health incidents in real time.
6. **Transportation.** Autonomous vehicles require and produce anywhere from 5 TB to 20 TB per day, gathering information about location, speed, vehicle condition, road conditions, traffic conditions and other vehicles. And the data must be aggregated and analyzed in real time, while the vehicle is in motion. This requires significant onboard computing -- each autonomous vehicle becomes an "edge." In addition, the data can help authorities and businesses manage vehicle fleets based on actual conditions on the ground.
7. **Retail.** Retail businesses can also produce enormous data volumes from surveillance, stock tracking, sales data and other real-time business details. Edge computing can help analyze this diverse data and identify business opportunities, such as an effective endcap or campaign, predict sales and optimize vendor ordering, and so on. Since retail businesses can vary dramatically in local environments, edge computing can be an effective solution for local processing at each store.

What are the benefits of edge computing?

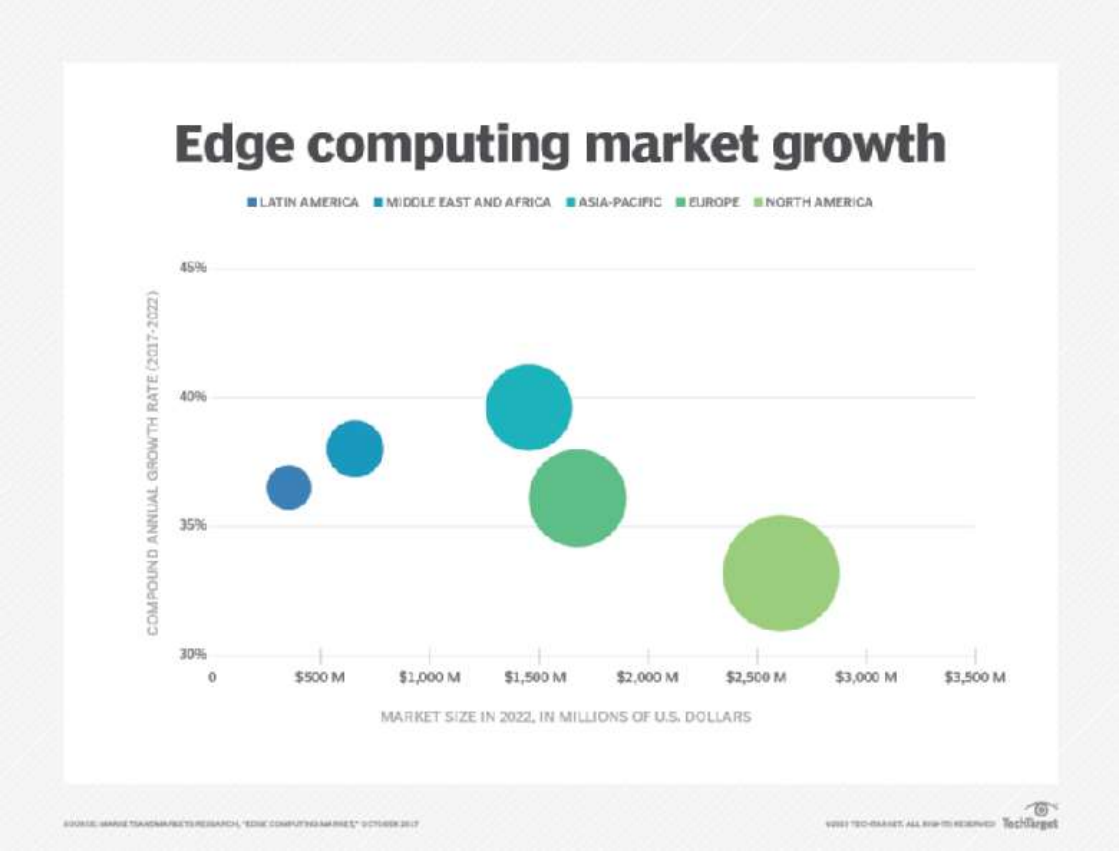
Edge computing addresses vital infrastructure challenges -- such as bandwidth limitations, excess latency and network congestion -- but there are several potential additional benefits to edge computing that can make the approach appealing in other situations.

Autonomy. Edge computing is useful where connectivity is unreliable or bandwidth is restricted because of the site's environmental characteristics. Examples include oil rigs, ships at sea, remote farms or other remote locations, such as a rainforest or desert. Edge computing does the compute work on site -- sometimes on the edge device itself -- such as water quality sensors on water purifiers in remote villages, and can save data to transmit to a central point only when connectivity is available. By processing data locally, the amount of data to be sent can be vastly reduced, requiring far less bandwidth or connectivity time than might otherwise be necessary.



Edge devices encompass a broad range of device types, including sensors, actuators and other endpoints, as well as IoT gateways.

Data sovereignty. Moving huge amounts of data isn't just a technical problem. Data's journey across national and regional boundaries can pose additional problems for data security, privacy and other legal issues. Edge computing can be used to keep data close to its source and within the bounds of prevailing data sovereignty laws, such as the European Union's GDPR, which defines how data should be stored, processed and exposed. This can allow raw data to be processed locally, obscuring or securing any sensitive data before sending anything to the cloud or primary data center, which can be in other jurisdictions.



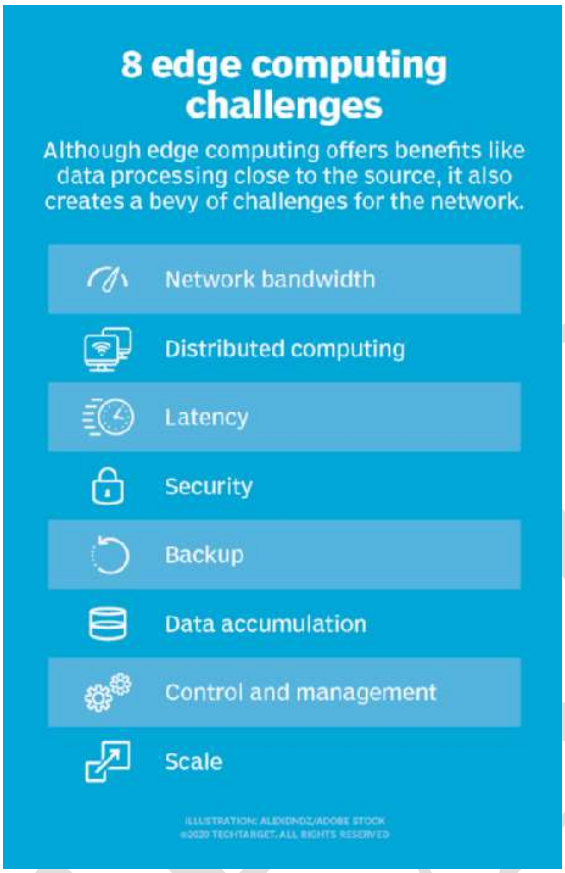
Research shows that the move toward edge computing will only increase over the next couple of years.

Edge security. Finally, edge computing offers an additional opportunity to implement and ensure data security. Although cloud providers have IoT services and specialize in complex analysis, enterprises remain concerned about the safety and security of data once it leaves the edge and travels back to the cloud or data center. By implementing computing at the edge, any data traversing the network back to the cloud or data center can be secured through encryption, and the edge deployment itself can be hardened against hackers and other malicious activities -- even when security on IoT devices remains limited.

Challenges of edge computing

Although edge computing has the potential to provide compelling benefits across a multitude of use cases, the technology is far from foolproof. Beyond the traditional problems of network limitations, there are several key considerations that can affect the adoption of edge computing:

- **Limited capability.** Part of the allure that cloud computing brings to edge -- or fog -- computing is the variety and scale of the resources and services. Deploying an infrastructure at the edge can be effective, but the scope and purpose of the edge deployment must be clearly defined -- even an extensive edge computing deployment serves a specific purpose at a pre-determined scale using limited resources and few services



- **Connectivity.** Edge computing overcomes typical network limitations, but even the most forgiving edge deployment will require some minimum level of connectivity. It's critical to design an edge deployment that accommodates poor or erratic connectivity and consider what happens at the edge when connectivity is lost. Autonomy, AI and graceful failure planning in the wake of connectivity problems are essential to successful edge computing.
- **Security.** IoT devices are notoriously insecure, so it's vital to design an edge computing deployment that will emphasize proper device management, such as policy-driven configuration enforcement, as well as security in the computing and storage resources -- including factors such as software patching and updates -- with special attention to encryption in the data at rest and in flight. IoT services from major cloud providers include secure communications, but this isn't automatic when building an edge site from scratch.
- **Data lifecycles.** The perennial problem with today's data glut is that so much of that data is unnecessary. Consider a medical monitoring device -- it's just the problem data that's critical, and there's little point in keeping days of normal patient data. Most of the data involved in real-time analytics is short-term data that isn't kept over the long term. A business must decide which data to keep and what to discard once analyses are performed. And the data that is retained must be protected in accordance with business and regulatory policies.
-

Edge computing implementation

Edge computing is a straightforward idea that might look easy on paper, but developing a cohesive strategy and implementing a sound deployment at the edge can be a challenging exercise.

The first vital element of any successful technology deployment is the creation of a meaningful business and technical edge strategy. Such a strategy isn't about picking vendors or gear. Instead, an edge strategy considers the need for edge computing. Understanding the "why" demands a clear understanding of the technical and business problems that the organization is trying to solve, such as overcoming network constraints and observing data sovereignty.

An edge data center requires careful upfront planning and migration strategies.

Such strategies might start with a discussion of just what the edge means, where it exists for the business and how it should benefit the organization. Edge strategies should also align with existing business plans and technology roadmaps. For example, if the business seeks to reduce its centralized data center footprint, then edge and other distributed computing technologies might align well.

As the project moves closer to implementation, it's important to evaluate hardware and software options carefully. There are many vendors in the edge computing space, including Adlink Technology, Cisco, Amazon, Dell EMC and HPE. Each product offering must be evaluated for cost, performance, features, interoperability and support. From a software perspective, tools should provide comprehensive visibility and control over the remote edge environment.

The actual deployment of an edge computing initiative can vary dramatically in scope and scale, ranging from some local computing gear in a battle-hardened enclosure atop a utility to a vast array of sensors feeding a high-bandwidth, low-latency network connection to the public cloud. No two edge deployments are the same. It's these variations that make edge strategy and planning so critical to edge project success.

An edge deployment demands comprehensive monitoring. Remember that it might be difficult -- or even impossible -- to get IT staff to the physical edge site, so edge deployments should be architected to provide resilience, fault-tolerance and self-healing capabilities. Monitoring tools must offer a clear overview of the remote deployment, enable easy provisioning and configuration, offer comprehensive alerting and reporting and maintain security of the installation and its data. Edge monitoring often involves an array of metrics and KPIs, such as site availability or uptime, network performance, storage capacity and utilization, and compute resources.

And no edge implementation would be complete without a careful consideration of edge maintenance:

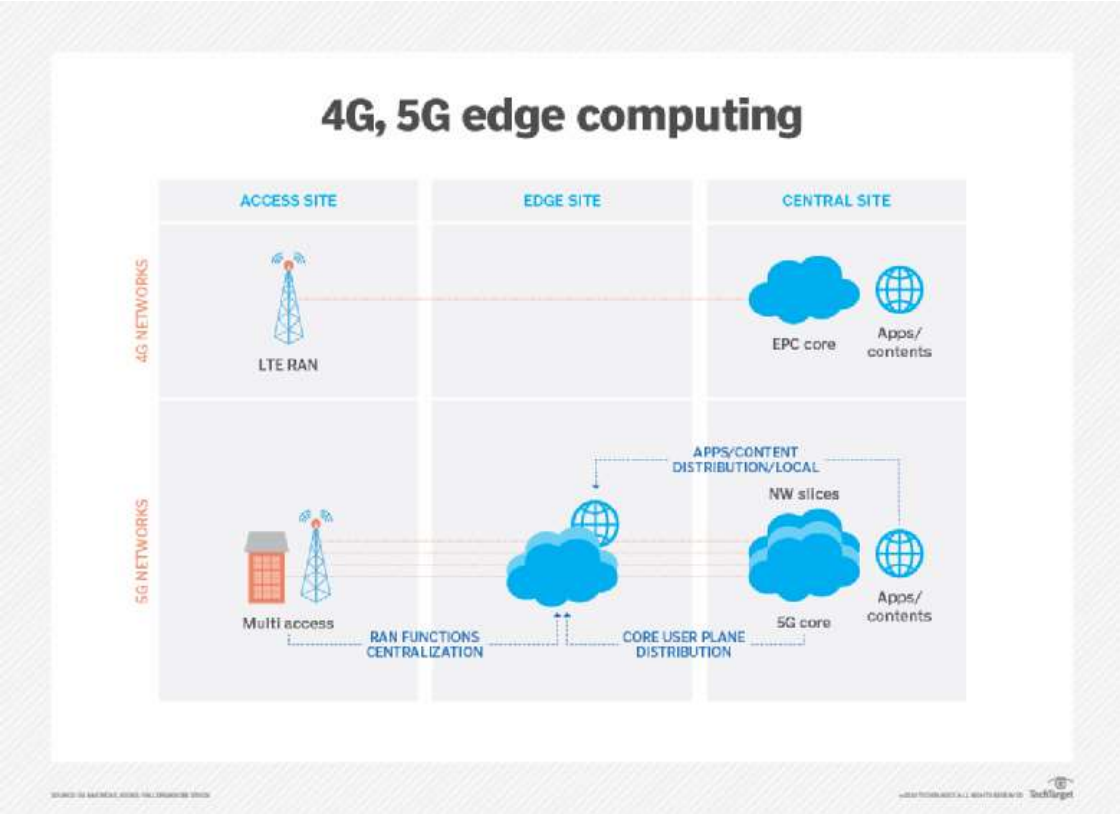
- **Security.** Physical and logical security precautions are vital and should involve tools that emphasize vulnerability management and intrusion detection and prevention. Security must extend to sensor and IoT devices, as every device is a network element that can be accessed or hacked -- presenting a bewildering number of possible attack surfaces.
- **Connectivity.** Connectivity is another issue, and provisions must be made for access to control and reporting even when connectivity for the actual data is unavailable. Some edge deployments use a secondary connection for backup connectivity and control.
- **Management.** The remote and often inhospitable locations of edge deployments make remote provisioning and management essential. IT managers must be able to see what's happening at the edge and be able to control the deployment when necessary.
- **Physical maintenance.** Physical maintenance requirements can't be overlooked. IoT devices often have limited lifespans with routine battery and device replacements. Gear fails and eventually requires maintenance and replacement. Practical site logistics must be included with maintenance.

Edge computing, IoT and 5G possibilities

Edge computing continues to evolve, using new technologies and practices to enhance its capabilities and performance. Perhaps the most noteworthy trend is edge availability, and edge services are expected to become available worldwide by 2028. Where edge computing is often situation-specific today, the technology is expected to become more ubiquitous and shift the way that the internet is used, bringing more abstraction and potential use cases for edge technology.

This can be seen in the proliferation of compute, storage and network appliance products specifically designed for edge computing. More multivendor partnerships will enable better product interoperability and flexibility at the edge. An example includes a partnership between AWS and Verizon to bring better connectivity to the edge.

Wireless communication technologies, such as 5G and Wi-Fi 6, will also affect edge deployments and utilization in the coming years, enabling virtualization and automation capabilities that have yet to be explored, such as better vehicle autonomy and workload migrations to the edge, while making wireless networks more flexible and cost-effective.



This diagram shows in detail about how 5G provides significant advancements for edge computing and core networks over 4G and LTE capabilities.

Edge computing gained notice with the rise of IoT and the sudden glut of data such devices produce. But with IoT technologies still in relative infancy, the evolution of IoT devices will also have an impact on the future development of edge computing. One example of such future alternatives is the development of micro modular data centers (MMDCs). The MMDC is basically a data center in a box, putting a complete data center within a small mobile system that can be deployed closer to data -- such as across a city or a region -- to get computing much closer to data without putting the edge at the data proper.