

SOFTWARE ENGINEERING [LECTURE NOTES]

Table of Contents

1. OVERVIEW	6
SOFTWARE ENGINEERING	8
SOFTWARE ENGINEERING AND THE WEB:	11
2. SOFTWARE PROCESS	12
ESSENTIAL ATTRIBUTES OF GOOD SOFTWARE	12
SOFTWARE PROCESSES	13
Software Process Models	13
Incremental development:	14
Reuse-oriented software engineering:	15
PROCESS ACTIVITIES	16
SOFTWARE DESIGN AND IMPLEMENTATION	17
SOFTWARE VALIDATION	18
Stages in the testing process:	18
SOFTWARE EVOLUTION	19
COPING WITH CHANGE	19
PROTOTYPING	20
INCREMENTAL DELIVERY	21
Boehm's spiral model	22
The Rational Unified Process	23
Phases of RUP:	24
Static workflows in the RUP:	24
3. AGILE SOFTWARE DEVELOPMENT	26
FUNDAMENTAL CHARACTERISTICS:	26
AGILE METHODS	26
Plan-driven and Agile Development	27
Extreme Programming	27
Characteristics / Practices	27
Extreme Programming Practices	27
TESTING PRACTICES	28
PAIR PROGRAMMING	28
AGILE PROJECT MANAGEMENT	28
Scaling Agile Methods	29
Critical adaptations:	30
4. REQUIREMENTS ENGINEERING	31
FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS	31
THE SOFTWARE REQUIREMENTS DOCUMENT	32
Structure of a requirements document:	32
Requirements Specification	33
NATURAL LANGUAGE SPECIFICATION	34
REQUIREMENTS ENGINEERING PROCESSES	34
Requirements Elicitation and Analysis:	35

SOFTWARE ENGINEERING

THE REQUIREMENTS ELICITATION AND ANALYSIS PROCESS:	35
Requirements Validation	37
REQUIREMENTS MANAGEMENT	38
5. SYSTEM MODELING.....	40
SYSTEM MODELS:.....	40
CONTEXT MODELS.....	40
UML activity diagrams:	40
Interaction models.....	41
Sequence diagrams.....	41
STRUCTURAL MODELS	42
Class diagrams	42
Generalization	42
BEHAVIORAL MODELS	42
Data-driven modeling	43
Event-driven modeling	43
Model-driven engineering	43
MODEL-DRIVEN ARCHITECTURE:	44
Types of abstract system models	44
6. ARCHITECTURAL DESIGN	46
Advantages of explicitly designing and documenting software architectures:.....	46
Architectural Design Decisions	46
Architectural pattern:	46
Architectural style:.....	46
Architectural views	47
Architectural Patterns.....	48
Model-View-Controller (MVC):.....	48
Application architectures.....	50
7. DESIGN AND IMPLEMENTATION	53
Object-oriented design using the UML.....	53
System context and interactions	53
Architectural design.....	53
Object class identification.....	54
Design models.....	54
Interface specification	55
Design patterns.....	55
Implementation issues	55
Configuration Management	56
Host-target Development.....	57
Open source Development.....	58
Open source Licensing.....	58
8. COMPONENT-BASED SOFTWARE ENGINEERING	59
Components and Component Models	60
Essential characteristics of a component as used in CBSE.	60
9. DISTRIBUTED SOFTWARE ENGINEERING.....	62

SOFTWARE ENGINEERING

Distributed Systems Issues	62
MODELS OF INTERACTION	65
PROCEDURAL INTERACTION	65
MESSAGE-BASED INTERACTION	66
Middleware.....	66
Client–server computing	67
Architectural patterns for distributed systems	68
Master-slave architectures	68
Two-tier client–server architectures	69
FAT Client Architecture for an ATM.....	70
Multi-tier client–server Architectures	70
Three Tier architecture for Internet Banking:	70
Use of Client-Server architectural patterns:.....	71
Distributed component architectures	72
Distributed Component Architecture for Data-Mining:	72
Distributed component architectures suffer from two major disadvantages:	73
Peer-to-peer architectures	73
P2P architectural models.....	73
A Decentralized P2P Architecture:	74
A Semi-Centralized Architecture:	75
SOFTWARE AS A SERVICE	75
Key elements of SaaS:.....	75
SaaS and SOA	75
Implementation factors for SaaS	75
Service configuration:.....	76
Multi-tenancy:	76
A multi-tenant database:.....	76
Scalability:.....	77
10. PLANNING A SOFTWARE PROJECT	78
Process Planning.....	78
Planning Guidelines	78
Effort Estimation.....	79
Top-Down Estimation Approach.....	79
The COCOMO Model	80
11. PROJECT SCHEDULING AND STAFFING.....	81
SOFTWARE CONFIGURATION MANAGEMENT PLAN	82
THE SCM PROCESS	83
Baseline Identification	84
Version Control	85
Local Version Control Systems.....	85
Centralized Version Control Systems	86
Distributed Version Control Systems.....	87
Change Control	88
Configuration Auditing	89
Formal technical reviews	89

SOFTWARE ENGINEERING

Reporting	89
Quality Plan.....	91
RISK MANAGEMENT	92
Risk Management Concepts	92
Risk Assessment.....	93
Risk Control.....	93
Project Monitoring Plan.....	94
Activity-level monitoring	94
Status reports	94
Milestone Analysis.....	94
12. SOFTWARE TESTING	95
TESTING FUNDAMENTALS	95
What is Software Testing?	95
Who does Software Testing?	95
Why is Software Testing important?	95
How to test the Software?.....	95
BLACK-BOX TESTING	95
Equivalence Class Partitioning.....	96
Boundary Value Analysis	96
Pairwise Testing.....	97
Special Cases.....	98
State-Based Testing	98
WHITE-BOX TESTING.....	99
Control flow testing.....	99
Data flow testing.....	100
Statement coverage.....	100
Decision coverage.....	101
Path testing.....	102
TESTING PROCESS.....	103
Test Plan	103
Test Case Design	104
Test Case Execution	105
DEFECT LIFE CYCLE	105
States in a Defect Life Cycle:.....	105

SOFTWARE ENGINEERING

1. Overview

Lots of people write programs.

- People in business write spreadsheet programs to simplify their jobs
- Scientists and engineers write programs to process their experimental data
- Hobbyists write programs for their own interest and enjoyment.

Majority of software development is:

- A professional activity where software is developed for specific business purposes
- For inclusion in other devices, or as software products such as information systems, CAD systems.
- for use by someone apart from its developer
- Is usually developed by teams rather than individuals.
- It is maintained and changed throughout its life.

Software engineering is intended to

- support professional software development, rather than individual programming
- Include techniques that support program specification, design, and evolution

Many people think that software is simply another word for computer programs.

However talking about software engineering:

- Software is not just the programs themselves but also all associated documentation and configuration data required to make these programs operate correctly.
- Is not a single program but a professionally developed system consisting of a number of separate programs and configuration files that are used to set up these programs.
- It may include system documentation, which describes the structure of the system; user documentation, which explains how to use the system, and websites for users to download recent product information.

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.

SOFTWARE ENGINEERING

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 engineers are concerned with developing software products which can be sold to a customer). There are two kinds of software products:

1. Generic products

- These are stand-alone systems that are produced by a development organization
- Sold on the open market to any customer who is able to buy them. Example: databases, word processors, drawing packages, and project-management tools.
- It also includes so-called vertical applications designed for some specific purpose such as library information systems, accounting systems, or systems for
- Maintaining dental records.
- the organization that develops the software controls the software specification

2. Customized products:

- Requested by a particular customer.
- A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.
- The specification is usually developed and controlled by the organization that is buying the software.

SOFTWARE ENGINEERING

Software Engineering

Definition: 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.

1. Engineering discipline:

- Engineers make things work.
- They apply theories, methods, and tools where these are appropriate.
- They use them selectively and always try to discover solutions to problems
- Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.

2. All aspects of software production:

- Software engineering is not just concerned with the technical processes of software development.
- It also includes activities such as software project management and the development of tools, methods, and theories to support software production.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project. For most types of systems, the majority of costs are the costs of changing the software after it has gone into use.

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

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

SOFTWARE ENGINEERING

Three general issues that affect many different types of software:

1. Heterogeneity

- Applicable to
 - Distributed systems across networks that include different types of computers and mobile devices.
 - All systems running on general-purpose computers
- Software may also have to execute on mobile phones.
- Integrate new software with older legacy systems written in different programming languages.
- Develop new techniques for building dependable software that is flexible enough to cope with this heterogeneity.

2. Business and social change

- Business and society are changing incredibly quickly as emerging economies require new technologies
- They need to change their existing software and to rapidly develop new software.
- Many traditional software engineering techniques are time consuming and delivery of new systems often takes longer than planned.
- The engineers need to evolve so that the time required for software to deliver value to its customers is reduced.

3. Security and trust

- As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface.
- We have to make sure that malicious users cannot attack our software and that information security is maintained.

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers.

How this systematic approach is actually implemented varies dramatically depending on the organization developing the software, the type of software, and the people involved in the development process.

There are no universal software engineering methods and techniques that are suitable for all systems and all companies.

1. 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. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.

2. Interactive transaction-based applications

These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. Obviously, these include web applications such as e-commerce applications where you can interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.

SOFTWARE ENGINEERING

3. 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. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car and software in a microwave oven to control the cooking process.

4. 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. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.

5. Entertainment systems

These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.

6. 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. These are often computationally intensive and require high-performance parallel systems for execution.

7. 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. The software has to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.

8. Systems of systems

These are systems that are composed of a number of other software systems. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

Software engineering fundamentals that apply to all types of software system:

1. The software should be developed using a managed and understood development process.
2. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed.
3. Different processes are used for different types of software.
4. Dependability and performance are important for all types of software systems.
5. Software should behave as expected, without failures and should be available for use when it is required.
6. The software should be safe and secure against external attack.
7. The system should perform efficiently and should not waste resources.
8. Deliver quality software within budget and schedule that will satisfy different types of customers.
9. Where appropriate, reuse existing software rather than write new software.

Software engineering and the Web:

The development of the World Wide Web has had a profound effect on our daily lives. Initially, the Web was primarily a universally accessible information store and it had little effect on software systems.

Around 2000, the Web started to evolve and more and more functionality was added to browsers. This meant that web-based systems could be developed where, instead of a special-purpose user interface, these systems could be accessed using a web browser.

The next stage in the development of web-based systems was the notion of web services. Web services are software components that deliver specific, useful functionality and which are accessed over the Web. Applications are constructed by integrating these web services, which may be provided by different companies.

In the last few years, the notion of 'software as a service' has been developed. It has been proposed that software will not normally run on local computers but will run on 'computing clouds' that are accessed over the Internet.

This radical change in software organization has, obviously, led to changes in the ways that web-based systems are engineered. For example:

1. Software reuse has become the dominant approach for constructing web-based systems.
2. When building these systems, pre-existing software components and systems are used to create new systems.
3. Web-based systems should be developed and delivered incrementally.
4. User interfaces are constrained by the capabilities of web browsers.

2. Software Process

Software Engineering is concerned with

- Technical processes of software development
- Software project management
- Development of tools, methods and theories to support software production
- Getting results of the required quality within the schedule and budget
- Often involves making compromises
- Often adopt a systematic and organized approach
- Less formal development is particularly appropriate for the development of web-based systems

Software Engineering is important because

- Individuals and society rely on advanced software systems
- Produce reliable and trustworthy systems economically and quickly
- Cheaper in the long run to use software engineering methods and techniques for software systems

Fundamental activities being common to all software processes:

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

Software Engineering is related to computer science and systems engineering:

- Computer science : Concerned with theories and methods
- Software Engineering: Practical problems of producing software
- Systems engineering: Aspects of development and evolution of complex systems. Specifying the system, defining its overall architecture, integrating the different parts to create the finished system

Essential attributes of good software

1. Maintainability

- Evolve to meet the changing needs of customers
- Software change is inevitable (see changing business environment)

2. Dependability and security

- Includes reliability, security and safety
- Should not cause physical or economic damage in case of system failure
- Take special care for malicious users

3. Efficiency

- Includes responsiveness, processing time, memory utilization
- Care about memory and processor cycles

4. Acceptability

5. Acceptable to the type of users for which it is designed

- Includes understandable, usable and compatible with other systems

SOFTWARE ENGINEERING

Application types

- Stand-alone applications
- Interactive transaction-based applications
- Embedded control systems
- Batch processing systems
- Entertainment systems
- Systems for modeling and simulation
- Data collection systems
- Systems of systems

Software engineering ethics:

- Confidentiality: respect confidentiality of employers and clients (whether or not a formal confidentiality agreement has been signed)
- Competence: do not misrepresent your level of competence (never accept work being outside of your competence)
- Intellectual property rights: be aware of local laws governing the use of intellectual property such as patents and copyright
- Computer misuse: do not use technical skills to misuse other people's computers

Software Processes

Main software processes

- Specification
- Design and implementation
- Validation
- Evolution

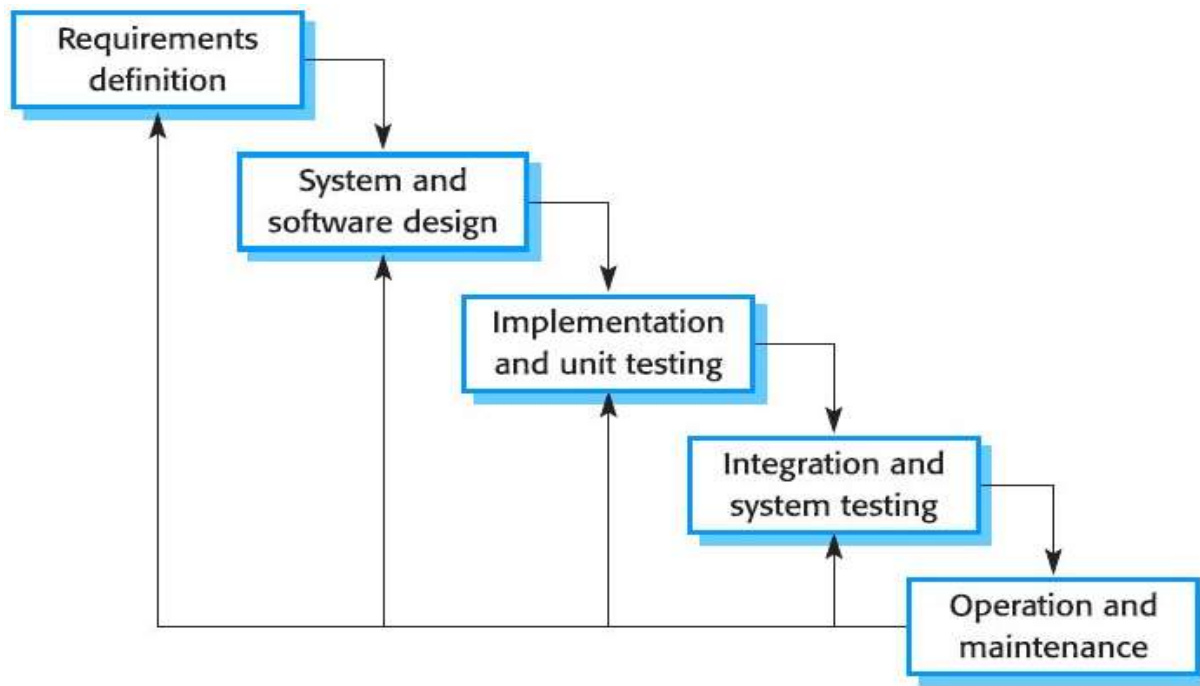
Software Process Models

- Waterfall model
- Incremental development
- Reuse-oriented software engineering

Waterfall model:

- 1. Requirements analysis and definition**
 - Consult system users to establish system's services, constraints and goals
 - They are then defined in detail and serve as a system specification
- 2. System and software design**
 - Establish an overall system architecture to allocate the requirements to either hardware or software systems
 - Involves identifying and describing the fundamental software system abstractions and their relationships
- 3. Implementation and unit testing**
 - Integrate and test individual program units or programs into complete systems
 - Ensure the software requirements have been met
 - Software system is delivered to the customer (after testing)
- 4. Operation and maintenance**
 - Normally the longest life cycle phase
 - System is installed and put into practical use
 - Involves correcting errors, improving the implementation of system units and enhancing the system's services as new requirements are discovered

SOFTWARE ENGINEERING



Incremental development:

1. Idea

- Develop and initial implementation
- Expose this to user comment
- Evolve this through several versions until an adequate system has been developed

2. Characteristics

- Start with an outline description of the software to develop
- Specification, development and validation activities are interleaved (concurrent activities) rather than separate
- More rapid feedback across activities
- Fundamental part of agile approaches
- Better for most business, e-commerce and personal systems than waterfall approaches

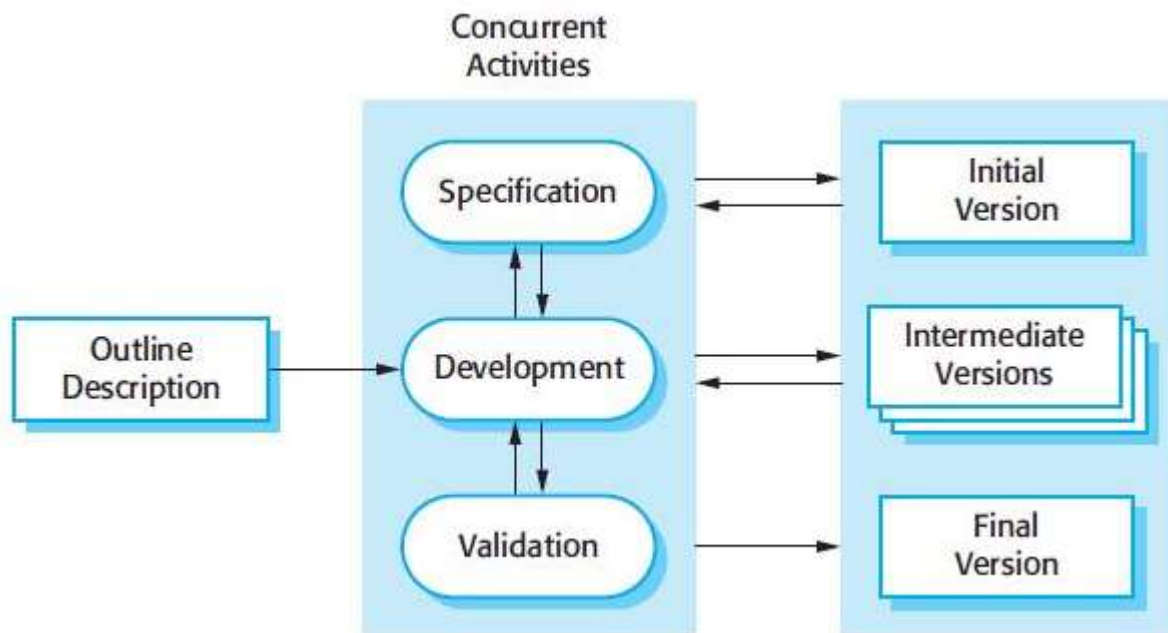
3. Benefits compared to waterfall model

- Reduced cost of accommodating changing customer requirements
- Easier to get customer feedback on the development work that has been done
- Possibly more rapid delivery and deployment of useful software to the customer. Customers use and gain value from the software earlier than with water model

4. Problems

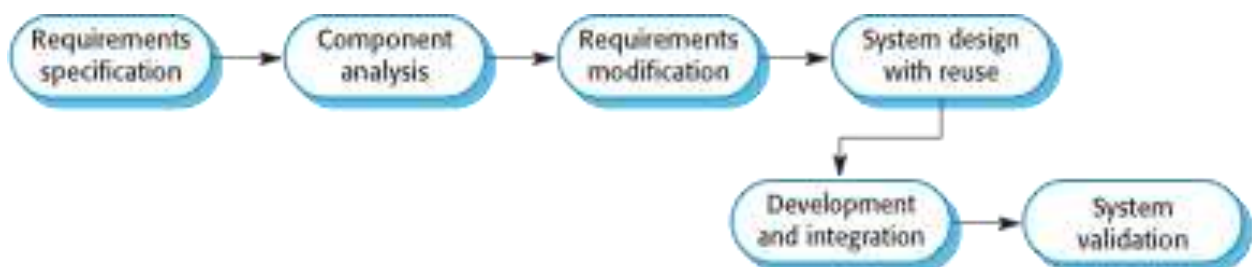
- **Process is not visible**
 - How to measure progress (see managers)
 - Producing documents that reflect every version of the system is not cost-effective
- **System structure tends to degrade as new increments are added**
 - Time and money is required on refactoring to improve the software
 - Regular changes tend to corrupt its structure
 - Further software changes become increasingly difficult and costly

SOFTWARE ENGINEERING



Reuse-oriented software engineering:

1. **Requirements specification**
2. **Component analysis**
 - Search for components to implement this specification
 - Usually exact matches are not found
3. **Requirements modification**
 - Analyze requirements using information about the found components
 - They are modified to reflect the available components
 - Restart with component analysis in case found components cannot be modified
4. **System design with reuse**
 - Design the framework of the system or reuse an existing framework
 - Some new software may have to be designed if reusable components are not available
5. **Development and integration**
 - Develop software that cannot be externally procured
 - Integrate components and COTS (commercial off-the-shelf systems)
 - System integration may be part of development process rather than a separate activity
6. **System validation**



SOFTWARE ENGINEERING

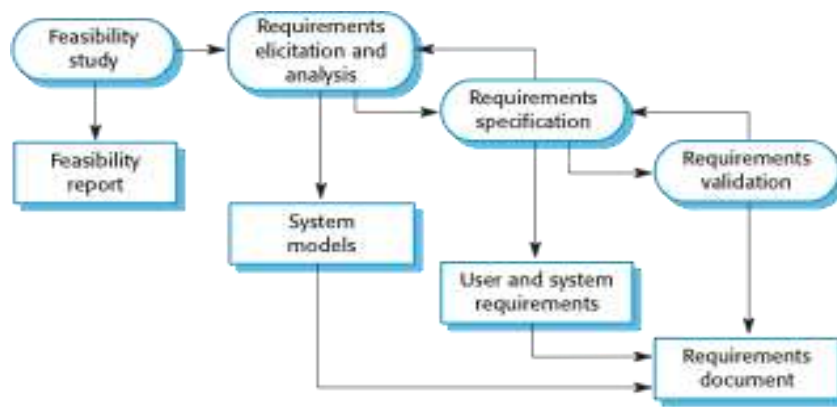
Types of software components to be used in a reuse-oriented process:

1. **Web services**
 - Developed according to service standards
 - Available for remote invocation
 - Collections of objects in a package (e.g. from .NET or J2EE)
 - Stand-alone software systems that are configured for use in a particular environment
2. **Advantages**
 - Reduce the amount of software to be developed
 - Reduce cost and risks
 - Usually leads to faster delivery of the software
3. **Disadvantages**
 - May lead to a system that does not meet the real needs of users (requirements compromises are inevitable)
 - Lost control over the system evolution as new versions of the reusable components are not under the control of the organization using them

Process Activities

1. **Software specification**
 - Definition: the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development
 - Requirements engineering: critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation
2. **Requirements engineering process**
 - i **Feasibility study**
 - Estimate whether the identified user needs may be satisfied using current software and hardware technologies
 - Considerations: cost-effective, development within existing budgetary constraints
 - Should be relatively cheap and quick
 - ii **Requirements elicitation and analysis**
 - Derive the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis
 - Development of one or more system models and prototypes possible
 - Help to understand the system to be specified
 - iii **Requirements specification**
 - Activity of translating the information gathered during the analysis activity into a document that defines a set of requirements
 - User requirements: abstract statements of the system requirements for the customer and end-user of the system
 - System requirements: more detailed description of the functionality to be provided
 - iv **Requirements validation**
 - Checks the requirements for realism, consistency and completeness
 - Errors are discovered to correct the specification

SOFTWARE ENGINEERING



Software design and implementation

Software implementation: process of converting a system specification into an executable system
Characteristics:

- Always involves processes of software design and programming
- May refine software specification if incremental approach is used

Software design: a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and the algorithms used.

Characteristics:

- Designers develop the design interactively
- Add formality and detail as they develop their design with constant backtracking to correct earlier designs
- Process activities are interleaved: feedback from one stage to another and consequent design rework
- Requires detailed knowledge about the software platform (environment in which the software will execute; e.g. operating system, database, middleware etc.)
- Activities in the design process vary, depending on the type of system being developed (see real-time systems vs. services vs. database systems)

Possible activities of the design process of information systems:

1. Architectural design

- Identify the overall structure of the system, the principal components (sub-systems, modules), their relationships and how they are distributed

2. Interface design

- Define the interfaces between system components
- Interface specification must be unambiguous (precise interface)
- After interface specifications are agreed, components can be designed and developed concurrently

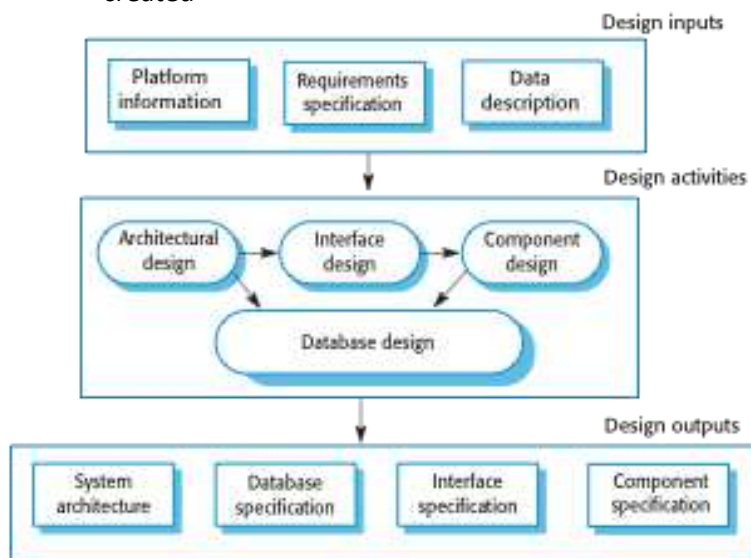
3. Component design

- Take each system component and design how it will operate
- May be a simple statement of the expected functionality to be implemented (specific design left to the programmer)
- May be a list of changes to be made to a reusable component or a detailed design model
- Design model may be used to automatically generate an implementation

SOFTWARE ENGINEERING

4. Database design

- Design the system data structures and how these are to be represented in a database
- Work depends on whether existing database is to be reused or a new database is to be created



Software validation

Definition: validation (verification and validation, V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the system customer

Stages in the testing process:

1. Development testing

- Components making up the system are tested by the people developing the system
- Individual/separate tests
- Test of functions, classes or coherent groupings of these entities
- Test automation tools (JUnit) are commonly used (when new versions of the components are created)

2. System testing

- System components are integrated to create a complete system
- Concerned with finding errors that result from unanticipated interactions between components and component interface problems
- Concerned with showing that the system meets its function and non-functional requirements
- Testing the emergent system properties
- May be a multi-stage process for large systems

3. Acceptance testing

- Final stage in the testing process before system is accepted for operational use
- Tested with data supplied by the customer rather than with simulated test data



Characteristics:

- Component development and testing processes are interleaved (normally)
- Incremental approach: each increment should be tested as it is developed
- Plan-driven approach: testing is driven by a set of test plans. Use of an independent team of testers
- Acceptance testing is sometimes called 'alpha testing'
- 'Beta testing' involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detect errors that may not have been anticipated by the system builders

Software Evolution

Distinction between software development and maintenance is increasingly irrelevant (software engineering as an evolutionary process where software is continually changed over its lifetime in response to changing requirements and customer needs)

Coping with Change

Reasons for change:

- Business procuring the system responds to external pressures and management priorities change
- New technologies become available
- New design and implementation possibilities emerge

Rework: work that has been completed has to be redone (software development, design, requirements analysis, redesign, re-test etc.)

Approaches to reduce cost of rework:**1. System prototyping**

- Version of the system or part of the system is developed quickly to check the customer's requirements and feasibility of some design decisions
- Supports change avoidance as it allows users to experiment with the system before delivery (refine their requirements)
- Reduced number requirements change proposals made after delivery

2. Incremental delivery

- Increments are delivered to the customer for comment and experimentation
- Supports both change avoidance and change tolerance
- Avoids premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low costs

Refactoring: improving structure and organization of a program (an important mechanism to support change tolerance)

Prototyping

Prototype: an initial version of a software system that is used to demonstrate concepts, try out design options and find out more about the problem and its possible solutions. Rapid, iterative development of the prototype is essential (to control costs; system stakeholders can experiment with the prototype early in the software process)

How software prototyping can help:

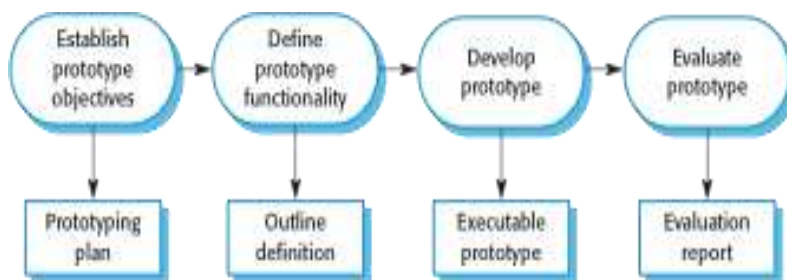
- Requirements engineering process: elicitation and validation of system requirements
- System design process: explore particular software solutions; support user interface design
- Get new ideas for requirements
- Find areas of strength and weakness in the software (to propose new system requirements)
- May reveal errors and omissions in the requirements
- Carry out design experiments to check the feasibility of a proposed design
- Essential part of the user interface design process (dynamic nature of user interfaces)

Delivering throwaway prototypes is usually unwise:

- May be impossible to tune the prototype to meet non-functional requirements: performance, security, robustness, reliability requirements
- Undocumented prototype due to rapid change during development; prototype code as the only design specification (bad for long-time maintenance)
- Probably degraded system structure due to changes made during prototype development. System will be difficult and expensive to maintain.
- Relaxed organizational quality standards for prototype development

Examples:

- Paper-based mock-ups of the system user interface (help users refine an interface design and work through usage scenarios)
- Wizard of Oz prototype (extension to the one above): only the user interface is developed



Incremental Delivery

Definition: an approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment

Characteristics:

1. Customers identify

- the services to be provided by the system (in outline)
- importance of the system's services to them

2. Processes

- Define outline requirements
- Assign requirements to increments
- Design system architecture
- Develop system increment
- Validate increment
- Integrate increment
- Validate system
- Deploy increment: if system is incomplete, restart with development of system increments
- Final system if system is complete

3. Advantages

- Customers can use early increments as prototypes and gain experience that informs their requirements for later system increments. No re-learning for users when system is available (part of the real system, unlike prototypes)
- Less waiting time for customers until entire system is delivered before they can gain value from it. Use the software immediately, which satisfies their most critical requirements
- Relatively easy to incorporate changes into the system (see incremental development)
- Most important system services receive the most testing (highest-priority services are delivered first). Customers less likely to encounter failures in the most important parts of the system

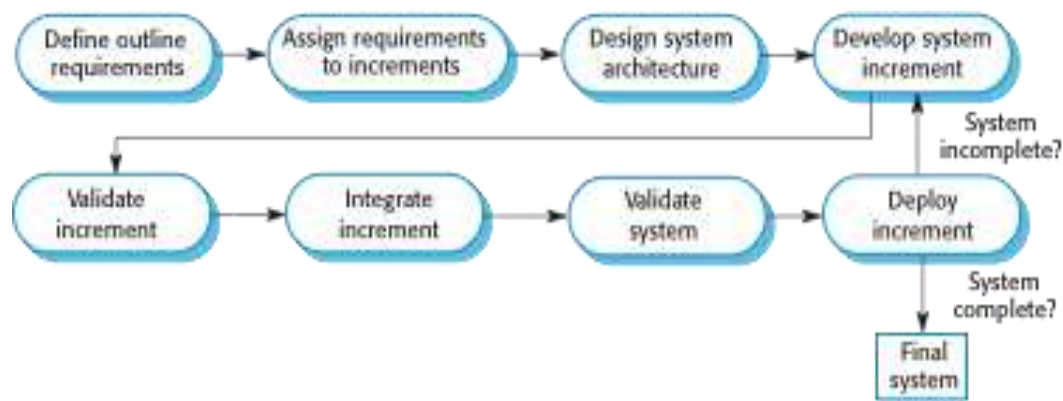
4. Problems with incremental delivery:

- Can be hard to identify common facilities that are needed by the increments (requirements are not defined in detail until an increment is to be implemented). Note that most systems require a set of basic facilities that are used by different parts of the system.
- Iterative development can also be difficult when a replacement system is being developed. Getting useful customer feedback is difficult (users want all the functionality of the old system and are often unwilling to experiment with an incomplete new system)
- Requires a new form of contract, which large customers (government agencies) may find difficult to accommodate
 - Specification is developed in conjunction with the software
 - No complete system specification until the final increment is specified
 - Conflicts with procurement model of many organizations: complete system specification is part of the system development contract!

5. Incremental development and delivery should not be used for (suffer from the same problems of uncertain and changing requirements)

- Very large systems where development may involve teams working in different locations
- Some embedded systems where software depends on hardware development
- Some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system

SOFTWARE ENGINEERING



Boehm's spiral model

- Is a risk-driven software process framework
- Software process is represented as a spiral
- Each loop in the spiral represents a phase of the software process
- It combines change avoidance with change tolerance
- It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks

Each loop is split into four sectors:

1. Objective setting

- To determine specific objectives, alternatives and constraints
- Constraints on the process and the product are identified
- A detailed management plan is drawn up
- Project risks are identified
- Alternative strategies may be planned

2. Risk assessment and reduction

- A detailed analysis for each identified project risk is carried out
- Measurements to reduce these risks

3. Development and validation

- A development model for the system is chosen
- Throwaway prototyping: user interface risks are dominant
- Development based on formal transformations: safety risks are the main consideration
- Waterfall model: main identified risk is a sub-system integration

4. Planning

- Project is reviewed
- Decision: continue with a further loop of the spiral? If so, plans are drawn up for the next phase of the project

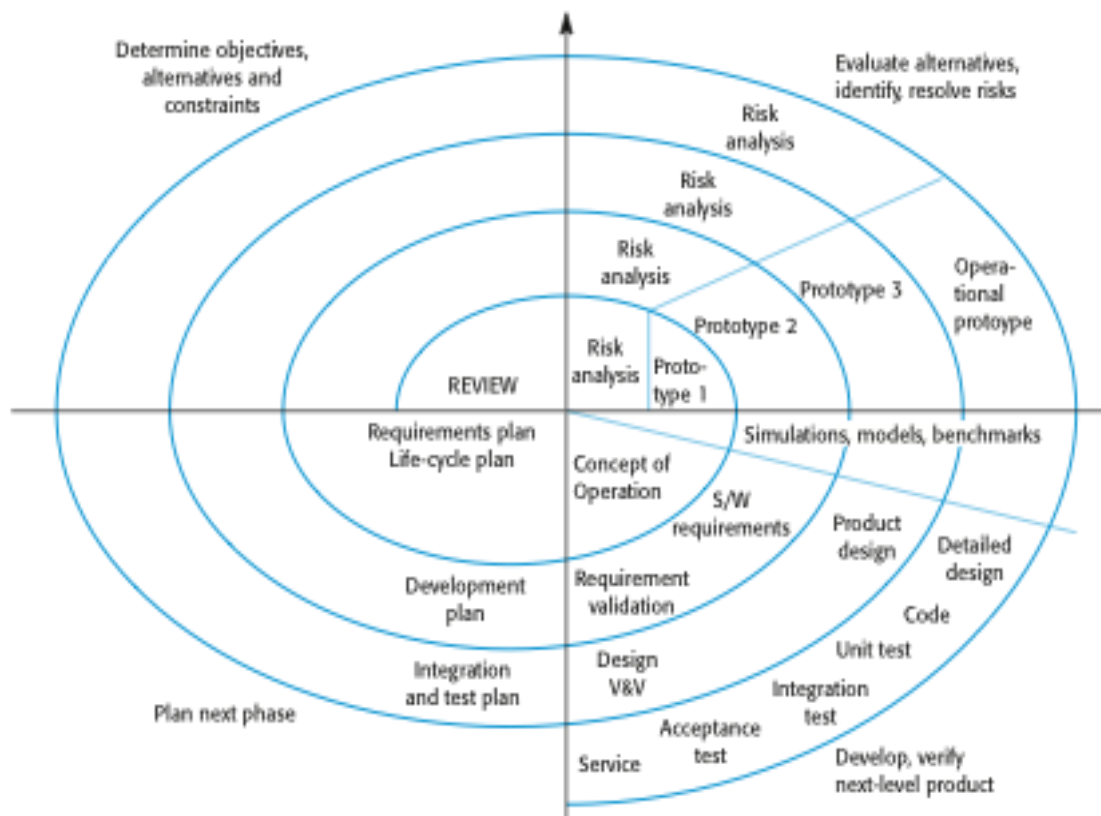
5. Advantages:

- Explicit recognition of risk

6. Characteristics:

- Begin of cycle of the spiral: elaborating objectives such as performance and functionality
- Enumerate alternative ways of achieving these objectives and dealing with the constraints
- Development is carried out after risks have been assessed

SOFTWARE ENGINEERING



The Rational Unified Process

1. A modern process model that has been derived from work on the UML and the associated Unified software Development Process
2. A hybrid process model
3. Brings together elements from all of the generic process models, illustrates good practice in specification and design and supports prototyping and incremental delivery
4. A phase model that identifies four discrete phases in the software process
5. Phases are more closely related to business rather than technical concerns (unlike the waterfall model where phases are equated with process activities)
6. Separation of phases and workflows
7. Recognizes that deploying software in a user's environment is part of the process
8. Phases are dynamic and have goals
9. Workflows are static and are technical activities that are not associated with a single phase (used throughout the development to achieve goals of each phase)

RUP described from three perspectives:

1. Dynamic perspective: shows the phases of the model over time
2. Static perspective: shows the process activities that are enacted
 - Called 'workflows' in the RUP description
 - 6 core process workflows and 3 core supporting workflows
3. Practice perspective: suggests good practices to be used during the process

SOFTWARE ENGINEERING

Phases of RUP:

1. Inception

- Goal: establish a business case for the system
- Identify all external entities (people and systems) that will interact with the system and define these interactions
- Assess the contribution that the system makes to the business
- Project may be cancelled after this phase if contribution is minor

2. Elaboration

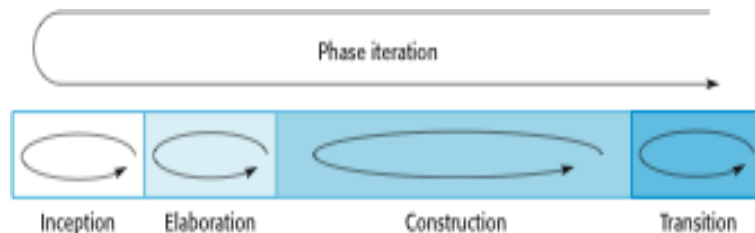
- Goal: develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan and identify key project risks
- On completion: requirements model for the system (may be a set of UML use-cases, an architectural description, and a development plan for the software)

3. Construction

- Involves system design, programming and testing
- Parts of the system developed in parallel and integrated during this phase
- On completion: working software system and associated documentation that is ready for delivery to users

4. Transition

- Moving the system from the development community to the user community and making it work in a real environment
- Sometimes ignored in most software process models
- An expensive and sometimes problematic activity
- On completion: a documented software system that is working correctly in its operational environment



Advantage of dynamic and static views:

Phases of the development process are not associated with specific workflows (all workflows may be active at all stages of the process)

Static workflows in the RUP:

1. Business modeling

- Business processes are modeled using business use cases Requirements
- Actors who interact with the system are identified
- Use cases are developed to model the system requirements

2. Analysis and design

- A design model is created and documented
- Used tools: architectural models, component models, object models and sequence models
- Implementation
- Components in the system are implemented and structured into implementation sub-systems
- Automatic code generation from design models helps accelerate this process

SOFTWARE ENGINEERING

3. Testing

- Carried out in conjunction with implementation (an iterative process)
- Follows the completion of the implementation

4. Deployment

- A product release is created, distributed to users and installed in their workplace
- Configuration and change management
- Manages changes to the system (supporting workflow)

5. Project management

- Manages the system development (supporting working)

6. Environment

- Makes appropriate software tools available to the software development team

Practice perspective and its recognized fundamental best practices (describes good software engineering practices):

- Develop software iteratively: plan increments of the system based on customer priorities
- Manage requirements: explicitly document the customer's requirements
- Use component-based architectures
- Visually model software: use UML models for static and dynamic views of the software
- Verify software quality
- Control changes to software

RUP is not suitable for

- Embedded software development

3. Agile software development

Fundamental Characteristics:

1. **Processes of specification, design and implementation are interleaved**
 - No detailed system specification
 - Design documentation is minimized or generated automatically by the programming environment
 - User requirements document only defines the most important characteristics of the system
2. **System is developed in a series of versions**
 - End-users and other system stakeholders are involved in specifying and evaluating each version
3. **System user interfaces are often developed using an interactive development system**
 - System may then generate a web-based interface for a browser or an interface for a specific platform
4. **Incremental development methods**
5. **Minimize documentation by using informal communications (rather than formal meetings with written documents)**

Agile Methods

1. **Typical types of system development:**
 - Small or medium-sized product for sale, developed by a software company
 - Custom system development within an organization,
 - with a clear commitment from the customer to become involved in the development process
 - without massive external rules and regulations
2. **Principles of agile methods:**
 - Customer involvement
 - Provide and prioritize new system requirements
 - Evaluate the iterations of the system
 - Incremental delivery
 - Customer specifies the requirements to be included in each increment
 - People not process
 - Recognize and exploit the skills of the development team
 - Team members should develop their own ways of working without prescriptive processes
 - Embrace change
 - Perspective: system requirements will change overtime
 - System design should accommodate these changes
 - Maintain simplicity
 - Simplicity for the software and for the development process
 - Actively work to eliminate complexity from the system
3. **Non-technical problems:**
 - Maintaining simplicity requires extra work
 - Prioritizing changes can be extremely difficult (esp. for stakeholders; see OpenGL)
 - Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods
 - Many organizations (large companies) are less likely to switch to more informal working environments (investigated a lot of time to structure and define processes)

SOFTWARE ENGINEERING

- Writing contracts between the customer and the supplier may be difficult, since software requirements documents are usually part of the contract

4. Questions when considering agile methods and maintenance:

- Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
- Can agile methods be used effectively for evolving a system in response to customer change requests?

Plan-driven and Agile Development

- Iteration occurs within activities with formal documents used to communicate between stages of the process
- Can support incremental development and delivery

Extreme Programming

- Several new versions of a system may be developed by different programmers, integrated and tested in a day
- Requirements are expressed as scenarios (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

Characteristics / Practices

- Incremental development by small, frequent releases of the system
- Customer involvement by continuous engagement of the customer in the development team
- Might avoid excessively long working hours by using pair programming, collective ownership of the system code and by sustainable development process
- Change is embraced through regular system releases to customers
- Maintaining simplicity by constant refactoring

Extreme Programming Practices

- Incremental planning
- Small releases
- Simple design
- Test-first development
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- Sustainable pace
- On-site customer

Testing Practices

1. Test-first development

- Writing the tests before writing the code
- Run the test as the code is being written and discover problems during development
- Reduces interface misunderstanding
- Task implementers have to thoroughly understand the specification to write tests for the system
- Avoids the problem of 'test-lag'

2. Incremental test development from scenarios

3. User involvement in the test development and validation

- Help develop acceptance tests for the stories that are to be implemented in the next release of the system
- Acceptance testing is incremental

4. The use of automated testing frameworks

- Essential for test-first development
- Tests are written as executable components before the task is implemented
- Should be stand-alone
- Should simulate the submission of input to be tested
- Widely used testing framework: Junit

Pair Programming

1. Advantages:

- Idea of collective ownership and responsibility for the system (egoless programming); team has collective responsibility for resolving these problems
- Informal review process: each line is looked at by at least two people
- Support refactoring (as a process of software improvement)
- Discuss software before development: probably have fewer false starts and less rework
- Less time spent repairing bugs by informal inspection

Agile Project Management

Scrum approach: a general agile method but with its focus on managing iterative development rather than specific technical approaches to agile software engineering

Phases:

1. Outline planning phase

- Establish the general objectives for the project
- Design the software architecture

2. Sprint cycles

- Each cycle develops an increment of the system
- Cycles: assess, select, review and develop

3. Project closure phase

- Wraps up the project
- Completes required documentation (system help frames and user manuals)

SOFTWARE ENGINEERING

- Assesses the lessons learned from the project

Key characteristics of sprint cycles:

- Sprints are fixed length (2-4 weeks)
- Starting point for planning: product backlog (list of work to be done on the project)
- Assessment phase: work is reviewed, priorities and risks are assigned
 - Customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each spring
- Selection phase: involves all of the project team
 - Select the features and functionality to be developed during the sprint
- Develop the software
 - Short daily meetings involving all team members to review progress and reprioritize work (if required)
 - Team is isolated from the customer and the organization
 - Communications are channeled through the 'Scrum master' (to protect the development team from external distractions)
 - No specific suggestions on how to write requirements, test-first development etc.
- End of sprint: work is reviewed and presented to stakeholders

Idea behind Scrum:

- Empower the whole team to make decisions
- Deliberately avoid the term 'project manager'
- 'Scrum master': 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

Advantages:

- Product is broken down into a set of manageable und understandable chunks
- Unstable requirements do not hold up progress
- Whole team has visibility of everything (improves team communication)
- Customers see on-time delivery of increments (gain feedback on how the product works)
- Established trust between customers and developers

Scaling Agile Methods

- Characteristics of large software system development:
- Collections of separate, communicating systems (in different places and time zones)
- 'brownfield systems': include and interact with a number of existing systems (political issues can be significant)
- Significant fraction of the development is concerned with system configuration rather than original code development (not necessarily compatible with incremental development and frequent system integration)
- Constrained by external rules and regulations limiting the way that they can be developed (system document required somehow)
- Long procurement and development time
- Diverse set of stakeholders (practically impossible to involve all of these different stakeholders in the development process)

SOFTWARE ENGINEERING

Two perspectives on the scaling of agile methods:

1. 'scaling up'

- Concerned with using these methods for developing large software systems that cannot be developed by a small team

2. 'scaling down'

- Concerned with how agile methods can be introduced across a large organization with many years of software development experience

Critical adaptations:

1. Up-front Design and System Documentation:

- software architecture has to be designed
- documentation to describe critical aspects of the system (database schemas, work breakdown across teams)

2. Cross-team Communication Mechanisms

- regular phone and video conferences
- short electronic meetings
- communication channels: emails, instant messaging, wikis, social networking systems

3. Continuous Integration

- practically impossible: whole system is build every time any developer checks in a change
- Essential: maintain frequent system builds and regular releases of the system
- New configuration management tools that support multi-team software development have to be introduced

Difficulties in introducing agile methods to large companies:

- Project managers: reluctant to accept the risk of a new approach (if they do not have experience of agile methods, and since they do not know how this will affect their particular projects)
- Large organizations: have quality procedures and standards for all of their projects
- A wide range of skills and abilities: agile methods seem to work best when team members have a relatively high skill level
- Cultural resistance to agile methods (organizations with long history of using conventional system engineering processes)

4. Requirements Engineering

1. **Requirements:** descriptions of what the system should do> the services that it provides and the constraints on its operation.
2. **Requirements Engineering (RE):** the process of finding out, analyzing, documenting and checking these services and constraints
3. **User requirements:** statements of what services the system is expected to provide to system users and the constraints under which it must operate, in a natural language plus diagrams
4. **System requirements:** more detailed descriptions of the software system's functions, services and operational constraints. The system requirements document / functional specification defines exactly what is to be implemented. It may be part of a contract between the system buyer and the software developers.

Functional and Non-functional Requirements

1. Functional requirements:

- Statements of services the system should provide
- How the system should react to particular inputs
- How the system should behave in particular situations
- Optional: what the system should do
Examples: security (user authentication facilities in the system)
- Vary from general to very specific requirements (what the system should do vs. local ways of working/organization of an existing system)
- May be expressed as user or system requirements or both
- Imprecisions may cause software engineering problems (see ambiguous user requirements)
- Function requirements specification should be complete and consistent (no contradictory definitions)
- Completeness and consistency practically impossible for large complex systems
- Easy to make mistakes and omissions when writing specifications for complex systems
- Many stakeholders in a large system (they often have inconsistent needs)

2. Non-functional requirements:

- Constraints on the services or functions offered by the system
- Include constraints concerning timing, development process and standards
- Often apply to the system as a whole (unlike individual features or services)
Example: security (limiting access to authorized users)
- May relate to emergent system properties: reliability, response time and store occupancy
- May define constraints on the system implementation: I/O device capabilities, data representations used in interfaces
- Specify or constrain characteristics of the system as a whole: performance security, availability
- Often more critical than individual function requirements
- Whole system may be unusable when failing to meet a non-functional requirement
- Often more difficult to relate components to nonfunctional requirements
- Implementation of these requirements may be diffused
- May affect the overall architecture of a system (rather than the individual components)
- May generate a number of related functional requirements that define new system services that are required (e.g. security)

SOFTWARE ENGINEERING

- May also generate requirements that restrict existing requirements
- Arise through user needs: budget constraints, organizational policies, need for interoperability with other software/hardware, external factors (safety regulations, privacy legislation)
- May come from
 - Product requirements: specify/constrain the behavior of the software
 - Usability requirements
 - Efficiency requirements (performance, space)
 - Dependability requirements
 - Security requirements
 - Organizational requirements: broad system req. derived from policies and procedures in the customer's and developer's organization.
 - Environmental requirement (operating environment of the system)
 - Operational process requirement (how the system will be used)
 - Development process requirement (programming language, development environment, process standards)
 - External requirements: from factor external to the system and its development process
 - Regulatory requirements (what must be done for the system to be used by a regulator)
 - Ethical requirements (ensure the system will be acceptable to its user and the general public)
 - Legislative requirements (ensure the system operates within the law)

Metrics for specifying non-functional requirements: speed, size, ease of use, reliability, robustness, portability

The Software Requirements Document

Synonyms: Software Requirements Specification, SRS

Definition: official statement of what the system developers should implement. It should include user and system requirements.

Diverse set of users:

- System customers who specify the req. and read them to check that they meet their needs (they also may make changes)
- (Senior) management of the organization that is paying for the system
- System Engineers responsible for developing the software
- System test engineers to develop validation tests for the system
- System maintenance engineers to understand the system and the relationships between its parts

Structure of a requirements document:

1. Preface

- Define the expected readership of the document
- Describe its version history
- A rationale for the creation of a new version
- A summary of the changes made in each version

2. Introduction

- Describe the need for the system

SOFTWARE ENGINEERING

- Describe the system's functions
 - Explain how it will work with other systems
 - Describe how the system fits into the overall business or strategic objectives of the organization commissioning the software
- 3. Glossary**
 - Define the technical terms used in the document
 - Should not make assumptions about the experience or expertise of the reader
 - 4. User requirements definition**
 - Describe the services provided for the user
 - Non-functional system requirements should also be described
 - May use natural language, diagrams or other notations that are understandable to customers
 - Product and process standards should be specified (if applicable)
 - 5. System architecture**
 - A high-level overview of the anticipated system architecture
 - Showing the distribution of functions across system modules
 - Architectural components should be highlighted (that are reused)
 - 6. System requirements specification**
 - Describe the functional and non-functional requirements in more detail
 - Optional: further detail to the non-functional requirements
 - Optional: interfaces to other systems
 - 7. System models**
 - Include graphical system models showing the relationships between the system components, the system and its environment. Examples: object models, data-flow models, semantic data models
 - 8. System evolution**
 - Describe the fundamental assumptions on which the system is based
 - Any anticipated changes due to hardware evolution, changing user needs etc.
 - Useful for system designers: may help to avoid design decisions
 - 9. Appendices**
 - Provide detailed, specific information that is related to the application being developed
 - Example: hardware and database descriptions
 - Hardware requirements: minimal and optional configurations for the system
 - Database requirements: logical organization of the data used by the system and the relationships between data
 - 10. Index**
 - A normal alphabetic index
 - An index of diagrams
 - An index of functions etc.

Requirements Specification

- Should be clear, unambiguous, easy to understand, complete, consistent
- Use of natural language, simple tables, forms and intuitive diagrams
- A complete and detailed specification of the whole system

How to write a System Requirement Specification:

- Natural language
- Structured natural language
- Design description languages
- Graphical notations
- Mathematical specifications

Natural Language Specification

Guidelines:

- Is expressive, intuitive and universal
- Use a standard format
- Use language consistently to distinguish between mandatory and desirable requirements
- Use text highlighting
- Do not assume that readers understand technical software engineering language
- Try to associate a rationale with each user requirement

Structures specifications:

- A way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way
- Maintains most of the expressiveness and understandability of natural language
- Ensures that some uniformity is imposed on the specification
- Define one or more standard templates for requirements and represent these templates as structured forms

Information in a standard form for specifying functional requirements:

- Description of the function or entity being specified
- Inputs and where these come from
- Outputs and where these go to
- Information about the information that is needed for the computation or other entities in the system that are used
- Action to be taken
- Pre- and post-condition before and after the function is called
- Side effects of the operation

Requirements Engineering Processes

1. High-level activities:

- Feasibility study
- Elicitation and analysis
- Specification
- Validation

2. Characteristics of a spiral view:

- Activities are organized as an iterative process around a spiral
- Output is the system requirements document
- Amount of time and effort for each activity depends on current stage and type of system to be developed

SOFTWARE ENGINEERING

Requirements engineering: the process of applying a structured analysis method (e.g. object-oriented analysis).

- Involves analyzing the system and developing a set of graphical system models (case models for system specification)
- Set of models describes the behavior of the system (annotated with additional information; e.g. system's required performance or reliability)

Requirements elicitation: a human-centered activity (unlike constraints by rigid system models)

Requirements management: process of managing changing requirements, e.g. modifications to the system's hardware, software or organizational environment.

Requirements Elicitation and Analysis:

- Software engineers work with customers and system end-users to find out about
 - the application domain
 - what services the system should the system provide
 - the required performance of the system
 - hardware constraints

The requirements elicitation and analysis process:

1. Requirements discovery

- Interact with stakeholders of the system to discover their requirements (domain requirements and documentation)
- Several complementary techniques available

2. Requirements classification and organization

- Takes unstructured collection of requirements
- Groups related requirements
- Organizes them into coherent clusters
- Most common way: model of the system architecture to identify sub-systems and to associate requirements with each sub-system
- Practice: requirements engineering and architectural design hard to separate

3. Requirements prioritization and negotiation

- Conflict of requirements with several stakeholders
- Prioritize requirements
- Find and resolve requirements conflicts through negotiation
- Compromises inevitable

4. Requirements specification

- Document requirements (formal or informal)

Eliciting and understanding requirements from stakeholders is difficult:

- Difficult to articulate
- Possibly unrealistic demands (stakeholders may don't know what is and isn't feasible)
- Stakeholders might not know what they want from a computer system (express in most general terms)
- Requirements engineers may not understand the naturally expressed requirements of stakeholders (in their own terms and with implicit knowledge)
- Different stakeholders have different requirements

SOFTWARE ENGINEERING

- Political factors
- Dynamic economic and business environment (where the analysis takes place)

Requirements discovery:

- Synonym: requirements elicitation
- The process of gathering information about the required system and existing systems
- Distilling the user and system requirements from this information
- Sources of information: documentation, system stakeholders, specifications of similar systems (plus interviews, observations)

Interviewing

- Formal and informal interviews
- Closed interviews: stakeholder answers a pre-defined set of questions
- Open interviews
 - No pre-defined agenda
 - Requirements engineering team explores a range of issues with system stakeholders
 - Develop a better understanding of their needs
- Practice: mixture of open and closed interviews
- Part of most requirements engineering processes
- Questions to stakeholders about the system that they currently use and the system to be developed
- Difficult to elicit domain knowledge through interviews (possibly)
 - Impossible to discuss domain requirements without using terminology. Inadequate knowledge leads to misunderstandings
 - 'basis'/fundamental (but professional) knowledge might be difficult to explain to non-professionals (in terms of software engineering or computer science)
- Ineffective technique for eliciting knowledge about organizational requirements and constraints
- Characteristics of effective interviews
 - Open-minded: avoid pre-conceived ideas about the requirements (surprising requirements)
 - Prompt the interviewee to get discussions (using a springboard questions, requirements proposal, b working together on a prototype system)
- Should be used with other requirements elicitation techniques (not to miss essential information)

Scenarios

- Often easier to relate to real-life examples rather than abstract descriptions
- Useful for adding detail to an outline requirements description
- Covers one or a small number of possible interactions
- Starts with an outline of the interaction
- May include
 - What the system and users expect when the scenario start
 - Normal flow of events in the scenario
 - What can go wrong and how this is handled
 - Information about other activities going on at the same time
- System state when the scenario finishes
- Involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios
- May be written as texts, supplemented by diagrams, screen shots etc.

SOFTWARE ENGINEERING

Use cases:

- A requirements discovery technique that were first introduced in the Object method
- A fundamental feature of the unified modeling language
- Identifies the actors involved in an interaction and names the type of interaction (supplemented by additional information describing the interaction with the system)
- Documented using a high-level use case diagram
- No hard and fast distinction between scenarios and use cases
- Identify the individual interactions between the system and its users or other systems
- Should be documented with a textual description (may be linked to other models in the UML)
- Effective technique (along with scenarios) for eliciting requirements form stakeholders who interact directly with the system
- UML is a de facto standard for object-oriented modeling

Ethnography:

- Software systems are used in social and organizational contexts (may constrain or influence software system requirements)
- Successful systems: satisfy social and organizational requirements
- Definition: an observational technique that can be used to understand operational processes and help derive support requirements for these processes
- Effective for discovering 2 types of requirements:
- Requirements that are derived from the way in which people actually work (rather than the way in which process definitions say they ought to work)
- Requirements that are derived from cooperation and awareness of other people's activities
- Can be combined with prototyping
- Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques)

Requirements Validation

- The process of checking that requirements actually define the system that the customer really wants
- Overlaps with analysis: it is concerned with finding problems with the requirements
- Important because errors in a requirements document can lead to extensive rework costs
- Process should not be underestimated

Checks during the validation process:

1. Validity checks

- Validity check is a verification performed, either through software or manually, to verify that no errors are present or that it adheres to a standard.

2. Consistency checks

- Requirements in the document should not conflict

3. Completeness checks

- Requirements that define all functions and constraints intended by the system user

4. Realism checks

- Check to ensure that requirements can actually be implemented

5. Verifiability

- System requirements should always be written so that they are verifiable
- Reduces the potential for dispute between customer and contract

SOFTWARE ENGINEERING

Requirements techniques to be used individually or in conjunction:

1. Requirements reviews

- Systematic analysis by a team of requires to check for errors and inconsistencies

2. Prototyping

- Demonstration of an executable model of the system
- Personal experimentation to check if it meets their real needs

3. Test-case generation

- Requirements should be testable
- Difficult or impossible design often means difficult implementation

Requirements Management

Definition: the process of understanding and controlling changes to system requirements

- Requirements for large systems are always changing. Reasons:
 - These systems are usually developed to address 'wicked' problems (software requirements are bound to be incomplete)
- Keep track of individual requirements
- Maintain links between dependent requirements
- Establish a formal process for making change proposals and linking these to system requirements
- Start planning how to manage changing requirements during the requirements elicitation process

Reasons why change is inevitable:

- Business and technical environment of the system always changes after installation
- People how pay for a system and the users of that system are rarely the same people
- Large systems usually have a diverse user community
 - Many users have different requirements and priorities that may be conflicting or contradictory
 - Final system requirements: a compromise

Requirements management planning:

- Establishes the level of requirements management detail that is required
- Required decisions to be made
 - Requirements identification: each requirements must be uniquely identified
 - A change management process: set of activities that assess the impact and cost of changes
 - Traceability policies: define the relationships between each requirement and between the requirements and the system design that should be recorded
 - Tool support: process large amounts of information about the requirements (specialist requirements management systems, spreadsheets, simple database systems)
- Required tool support for
 - Requirements storage: requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process
 - Change management: process of change management is simplified if active tool support is available

SOFTWARE ENGINEERING

- Traceability management: some tools help discover possible relationships between requirements

Requirements change management:

- Should be applied to all proposed changes to a system's requirements (after they have been approved)
- Essential because
 - Need to decide if the benefits of implementing new requirements are justified by the costs of implementation
- Advantage of a formal change management
 - Change proposals are treated consistently
 - Changes to the requirements document are made in a controlled way

Principal stages to a change management process:

- Problem analysis and change specification
- Change analysis and costing
- Change implementation

5. System Modeling

Definition: the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

- Represent the system through graphical notation (UML) or formal models (mathematics)
- Used during requirements engineering process: help derive the requirements for a system
- Used during design process: describe the system to engineers implementing the system and after implementation to document the systems structure and operation

System models:

- Abstraction of the system; leaves out details

1. Perspectives for creating models:

- External perspective: model the context/environment of the system
- Interaction perspective: model the interactions between a system and its environment or between the components of a system
- Structural perspective: model the organization of a system or the structure of the data
- Behavioral perspective: model the dynamic behavior of the system and how it responds to events

2. Type of diagrams representing the essentials of a system:

- Activity diagrams: activates involved in a process or in data processing
- Case diagrams: interactions between a system and its environment
- Sequence diagrams: interactions between actors and the system and between system components
- Class diagrams: object classes in the system and the associations between these classes
- State diagrams: how the system reactions to internal and external events

3. Why graphical models are used:

- A means of facilitating discussions about an existing or proposed system
- A way of documenting an existing system
- A detailed system description that can be used to generate a system implementation

Context models

- Normally show that the environment includes several other automated systems
- Do not show types of relationships between the systems in the environment and the system to be specified
- Used along with other models (business process models)

UML activity diagrams:

- Show the activities that make up a system process
- Show the flow of control from one activity to another
- Start of process: indicated by a filled circle
- End of process: filled circle inside another circle
- Activities: rectangles with round corners
- Arrows represent the flow of work from one activity to another
- Arrows annotated with guards: indicate the condition when that flow is taken
- Solid bar: indicate activity coordination

Interaction models

Some types of interaction:

- User interaction: user inputs and outputs
- System interaction: systems and components

Modeling user interaction:

- Helps to understand if a proposed system structure is likely to deliver the required system performance and dependability

Use case modeling:

- Used to model interactions between a system and external actors (users, other systems)
- Used to support requirements elicitation
- Gives a fairly simple overview of an interaction (you have to provide more detail to understand what is involved: textual description, structured description in a table, sequence diagram)

Use case diagrams

- Represents a discrete task (involves external interaction with a system)
- Shown as an ellipse with the actors involved in the use case

Key elements often involved in use cases: actors, description, data, stimulus, response, comments

Sequence diagrams

- Used to model the interactions between
 - the actors and the objects in a system
 - the objects themselves
- Shows the sequence of interactions that take place during a particular use case or use case instance
- Objects and actors involved are listed along the top of the diagram
- Interactions between objects are indicated by annotated arrows
- Rectangle on the dotted lines indicates the lifeline of the object concerned
- Read the sequence of interactions from top to bottom
- Annotations on the arrows indicate the calls to the objects, their parameters and the return values
- Include every interaction in sequence diagrams for code generation or detailed documentation

Structural models

- Display the organization of a system in terms of the components that make up that system and their relationships
- Static models: show the structure of the system design
- Dynamic models: show the organization of the system when it is executing
 - May be very different from a static model of the system components (for a dynamic organization of a system as a set of interacting threads)
- Creation of structural models:
 - Through discussions and system architecture designs

Class diagrams

- Used when developing an object-oriented system model to show the classes in a system and the associations between these classes
- Association: a link between classes that indicates that there is a relationship between these classes
 - Each class may have some knowledge of its associated class
- Expressed at different levels of detail
 - World: develop of model (essential objects)
- Types of relationship: 1 to 1, m to 1, m to n

Generalization

- An everyday technique that we use to manage complexity
- Easier to identify common characteristics
- Possible to make general statements that apply to all class members
- UML has a specific type of association to denote generalization
 - Shown as an arrowhead pointing up to the more general class
 - Attributes and operations associated with higher-level classes are also associated with the lower-level classes
 - Lower-level classes are subclasses inherit the attributes and operations from their super classes
 - Lower-level classes add more specific attributes and operations

Aggregation:

- UML provides a special type of association between classes called aggregation
- One object is composed of other objects
- Use a diamond shape next to the class that represents the whole

Behavioral models

- Show what happens or what is supposed to happen when a system responds to a stimulus from its environment
 - Data: data arrives that has to be processed by the system
 - Events: event happens that triggers system processing (may have associated data)
- Dynamic behavior of the system (as it is executing)

Data-driven modeling

- Show the sequence of actions involved in processing input data and generating an associated output
- Useful during the analysis of requirements as they can be used to show end-to-end processing in a system
- Data-flow models:
 - Illustrate processing steps in a system
 - Tracking and documenting how the data associated with a particular process moves through the system
 - Helps analysts and designers understand what is going on
 - Simple and intuitive
 - Possible to explain them to potential system users who can then participate in validating the model
 - Focus on system functions
 - Do not recognize system objects (therefore, data flow models not included in UML)
 - Similar to activity diagrams in UML 2.0 (data-driven systems are common in business)
 - Highlight functions
- Alternate way: use UML sequence diagrams
 - Highlight objects in a system

Event-driven modeling

- Shows how a system responds to external and internal events
- Based on the assumption that
 - A system has a finite number of states
 - Events may cause a transition from one state to another
- Particularly appropriate for real-time systems
- State diagrams (UML):
 - Show system states and events that cause transitions from one state to another
 - Do not show the flow of data within the system
 - May include additional information on the computations carried out in each state
 - Rounded rectangles: represent system states (may include a brief description of the actions taken in that state)
 - Labeled arrows: represent stimuli that force a transition
 - Optional: start and end states using filled circles (see activity diagrams)

Model-driven engineering

- An approach to software development where models rather than programs are the principal outputs of the development process
- Programs (execute on hardware/software platform) are generated automatically from models
- Raises level of abstraction in software engineering (no longer have to concerned with programming language details or specifics of execution platforms)
- Roots in model-driven architecture (MDA); was proposed by the Object Management Group (OMG)
- Concerned with all aspects of the software engineering process (including model-based requirements engineering, software processes for model-based development, model-based testing)

SOFTWARE ENGINEERING

Advantages:

- Allows engineers to think about systems at a high level of abstraction
- No concern for the details of their implementation
- Reduces the likelihood of errors
- Speeds up the design and implementation process
- Allows for the creation of reusable, platform-independent application models
- Only a translator for a platform is required to adapt the system to some new platform technology

Disadvantages:

- Abstractions that are supported by the models are not always the right abstractions for implementations (using an off-the-shelf, configurable package)
- Arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system's lifetime (for this class of systems, implementation is not the major problem)
- Requirements engineering, security and dependability, integration with legacy systems, and testing are more significant
- Not clear whether the costs and risks of model-driven approaches outweigh the possible benefits

Model-driven architecture:

- Is a model-focused approach to software design and implementation
- Uses a sub-set of UML models to describe the system
- Models at different levels of abstraction are created
- Generate a working program without manual intervention from a high-level platform independent model (in principle)
- Complete automated translation tools are unlikely to be available in the near future
- Translation of PIMs to PSMs is more mature (several commercial tools exists)
- An uneasy relationship between agile methods and model-driven architecture

Types of abstract system models

- **Computation independent model (CIM):**
 - Models the important domain abstractions used in the system
 - Sometimes called domain models
- **Platform independent model (PIM):**
 - Models the operation of the system without reference to its implementation
 - Usually described using UML models that show the static system structure and how it responds to external and internal events
- **Platform specific models (PSM):**
 - Transformations of the platform-independent model with a separate PSM for each application platform
 - Layers of PSM may add some platform-specific detail
 - First-level PSM: middleware-specific, database independent

Executable UML:

- Have to be able to construct graphical models whose semantics are well defined
- Need to add information to graphical models about the ways in which the operations defined in the model are implemented
- Supported by xUML or Executable UML
- UML is more concerned with software design and expressiveness, rather than programming language or semantic details of the language
- Number of model types has been dramatically reduced to 3 key model types (to create executable sub-set of UML):
 - Domain models: identify the principal concerns in the system (UML class diagrams with objects, attributes and associations)
 - Class models: classes are defined (along with their attributes and operations)
 - State models: a state diagram is associated with each class and is used to describe the lifecycle of the class
- Dynamic behavior of the system may be
 - Specified declaratively using the object constraint language (OCL)
 - Expressed using UMLs action language (a very high-level programming language)

6. Architectural Design

- Concerned with understanding how a system should be organized and designing the overall structure of that system
- The first stage in the software design process
- The critical link between design and requirements engineering (identifies the main structural components in a system and the relationships between them)
- Output: architectural model (describes how the system is organized as a set of communicating components)
- A significant overlap between the processes of requirements engineering and architectural design (in practice) (a system specification should not include any design information)
- Different levels of abstraction
 - Individual program architectures
 - Distributed system architectures (composed of several programs, components)
- Affects performance, robustness, disreputability, maintainability of a system
- Modeled using
 - Simple block diagrams (a box represents a component)
 - Arrows (signals or data flow between components)

Advantages of explicitly designing and documenting software architectures:

- Stakeholder communication (high-level presentation; focus discussions)
- System analysis (effect on design's critical requirements: performance, reliability, maintainability)
- Large-scale reuse (system architecture often the same for systems with similar requirements)

Ways an architectural model is used:

- Facilitate discussions about the system design
- Document an designed architecture

Architectural Design Decisions

- A creative process: design a system organization that will satisfy the functional and non-functional requirements of a system
- Activities within the process depend on the
 - Type of system being developed
 - Background and experience of the system architect
 - Specific requirements for the system
- A series of decisions to be made rather than a sequence of activities
- Architecture may be based on a particular pattern or style

Architectural pattern:

- A description of a system organization (client-server organization, layered organization)
- Capture the essence of an architecture that has been used in different software systems

Architectural style:

- Choose most appropriate structure (client-server, layered)

SOFTWARE ENGINEERING

Architectural pattern and style should depend on the non-functional system requirements:

1. Performance:

- Localize critical operations within a small number of components
- Deployed on the same computer (rather than distributed)
- Use a few relatively large components and fine-grain components
- Reduce component communications
- Consider run-time system organizations (allows the system to be replicated and executed on different processors)

2. Security:

- Layered structure should be used
- Most critical assets protected in the innermost layers
- A high level of security validation applied to these layers

3. Safety:

- Safety-related operations should be located in either a simple components or in a small number of components
- Reduces the cost and problems of safety validation
- Provides related protection systems (safely shut down system if failure)

4. Availability:

- Include redundant components (replace and update components without stopping the system)

5. Maintainability:

- Use fine-grain, self-contained components
- Separate producers of data from consumers
- Shared data structures should be avoided

Architectural views

Impossible to represent all relevant information about a system's architecture in a single architectural model. Views might be:

- How the system is decomposed into modules
- How the run-time processes interact

1. Logical view

- Key abstractions in the system (as objects, object classes)
- Relate system requirements to entities in this logical view

2. Process view

- How system is composed of interacting processes (at run-time)
- Useful for making judgments about non-functional system characteristics (performance, availability)

3. Development view

- How the software is decomposed for development
- Shows the breakdown of the software into components (implemented by a single developer or developer team)
- Useful for software managers and programmers

SOFTWARE ENGINEERING

4. Physical view

- How software components are distributed across the processors in the system
- Shows system hardware
- Useful for system engineers planning a system deployment

5. Conceptual view

- Abstract view of the system
- Bases for decomposing high-level requirements into more detailed specifications
- Help engineers make decisions about components (that can be reused)
- Represent a product line rather than a single system
- Used to support architectural decision making
- A way of communicating the essence of a system to different stakeholders
- Possible to associate architectural patterns with different view of a system

6. Languages to describe views

- UML (might remain most commonly used way of documenting system architectures)
- ADLs (specialized architectural description languages) (components and connectors, rules, guidelines)

Architectural Patterns

- Patterns as a way of presenting, sharing and reusing knowledge about software system is widely used
- Patterns for organizational design, usability, interaction, configuration management
- A stylized, abstract description of good practice (tried and testing in different systems and environments)
- Describe a system organization that has been successful in previous systems
- Include information of when it is and is not appropriate to use that pattern, the pattern's strengths and weaknesses

Model-View-Controller (MVC):

- Basis of interaction management in many web-based systems

Layered architecture

- Separation and independence allow changes to be localized
- System functionality: organized into separate layers
- Each layer relies on the facilities and services offered by the layer immediately beneath it
- Supports incremental development of systems
- Architecture: changeable and portable (layers can be easily replaced as long as the interface is constant)
- Easier to provide multi-platform implementations of an application system (machine dependencies in inner layers)
- Disadvantages:
 - Clean separation between layers often difficult
 - High-level layer may have to interact directly with lower-level layers (rather than through the layers immediately below)
 - Performance penalty (multiple levels of interpretation of a service request)
- Sample: a generic layered architecture
 - User interface
 - User interface management; authentication and authorization
 - Core business logic; application functionality; system utilities
 - System support (OS, database etc.)

SOFTWARE ENGINEERING

- Sample: LIBSYS
 - 5 layers
 - Allows controlled electronic access to copyright material from a group of university libraries

Repository Architecture:

- All data in a system is managed in a central repository (accessible to all system components)
- Components do not interact directly (through repository instead)
- When to use
 - System with large volumes of information generated (stored for a long time)
 - Data-driven systems (inclusion of data in the repository triggers an action or tool)

Advantages:

- Components can be independent
- Changes made by one component can be propagated to all components
- All data can be managed consistently (in one place)

Disadvantages:

- Single point of failure: problems in the repository affect the whole system
- Inefficiencies in organizing all communication through the repository
- Difficult to distribute the repository across several computers

Sample: architecture for an IDE:

- UML editors
- Code generators
- Java editor
- Python editor
- Report generator
- Design analyzer
- Design translator

Client-server architecture:

- System's functionality is organized into services (each services delivered from a separate server)
- Clients: users of these services; they access servers to make use of them

When to use:

- Data in a shared database has to be accessed from a range of locations
- Load on a system is variable (servers can be replicated)

Advantages:

- Servers can be distributed across a network
- General functionality can be available to all clients (does not need to implemented by all services)

Disadvantages:

- Each service is a single point of failure
- Susceptible to denial of service attacks or server failure
- Performance: unpredictable (depends on the network and system)
- Management problems possible (servers owned by a different organization)
- Sample: video/DVD library

SOFTWARE ENGINEERING

- Catalogue server
- Video server
- Picture server
- Web server

Pipe and filter architecture:

- Run-time organization of a system
- Functional transformations process their inputs and produce outputs
- Data flows from one to another and is transformed as it moves through the sequence
- Each processing component (filter) is discrete (carries out one type of data transformation)
- When to use
 - Data processing applications (batch- and transaction-based)
 - Applications where inputs are processed in separate stages to generate related outputs

Advantages:

- Easy to understand
- Supports transformation reuse
- Workflow style matches structure of many business processes
- Evolution by adding transformation is straightforward
- Implemented as sequential or concurrent system

Disadvantages:

- Format of data transfer has to be agreed upon between communicating transformations
- Each transformation must parse input and unparsed output to the agreed form
- Increases system overhead
- Possibly impossible to reuse functional transformations that use incompatible data structures

Application architectures

- Intended to meet a business or organizational need
- Business characteristics: Hire people, issue invoices, keep accounts etc.
- Business operating: use common sector-specific applications
- Encapsulate the principal characteristics of a class of systems
- May be re-implemented when developing new systems
- Possible application reuse without reimplementation for many business systems (ERP, SAP, Oracle, COTS)
- Identify stable structural features of the system architectures

Ways of using models of application architectures:

- As a starting point for the architectural design process
 - If unfamiliar with type of application
 - Base initial design on a generic application architecture (specialization required)
- As a design checklist
 - Compare developed architectural design with generic application architecture
 - Check consistency
- As a way of organizing the work of the development team
 - Possible to develop application architectures in parallel
 - Assign work to group members to implement components within the architecture
- As a means of assessing components for reuse
 - Compare available components with generic structures

SOFTWARE ENGINEERING

- Comparable components in the application architecture?
- As a vocabulary for talking about types of applications
 - Use concepts identified in the generic architecture to talk about the applications

Transaction processing systems:

- Abbreviation: TP systems
- Designed to process user requests for information from a database, or requests to updated a database
- Database transaction: a sequence of operations that is treated as a single unit (an atomic unit) (and all of the operations have to be completed before the database changes are made permanent)
- Prevent inconsistencies in the database
- Transaction: a coherent sequence of operations that satisfies a goal
- TP systems are usually interactive systems (users make asynchronous requests for services)
- Possible organization: pipe and filter architecture
- Sample: ATM system

Information systems:

- Allows controlled access to a large base of information
- Samples: library catalog, flight timetable, records of patients in a hospital
- Increasingly web-based systems
- Possible implementation: layered architecture
 - Top layer: user interface (UI)
 - Second layer: UI functionality (delivered through web browser) (log in, checking components, data validation components, menu management components)
 - Third layer: functionality of the system (components of system security etc.)
 - Lowest layer: database management system
- Often implemented as multi-tier client-server architectures
 - Web server: responsible for all user communications
 - Application server: application-specific logic, information storage, information retrieval request
 - Database server: moves information to/from database, handles transaction management

Language processing systems:

- Translate natural/artificial languages into another representation of that language
- May execute resulting code for programming languages (e.g. script languages)
- Samples:
 - Compilers translate a programming language into machine code
 - Translate an XML data description into commands to query a database to an alternative XML representation
 - Natural language processing systems may translate one natural language to another
- Possible architecture (programming language)
 - Source language instructions: define the program to be executed
 - Translator: converts source language instructions into instructions for an abstract machine
 - Interpreter: fetches the instructions for execution; executes them using data from the environment (if required) (usually hardware unit processing machine instructions or software component)
 - Output of the process: result of interpreting the instructions on the input data

Pipe and filter compiler architecture (components):

1. Lexical analyzer

- Takes input language tokens
- Converts them to an internal form

2. Symbol table

- o Holds information about the names of entities (variables, class names, object names etc.)

3. Syntax analyzer

- o Checks syntax of the language being translated
- o Uses a defined grammar of the language
- o Builds a syntax tree

4. Syntax tree

- o An internal structure representing the program being compiled

5. Semantic analyzer

- o Uses information from the syntax tree and symbol table
- o Checks semantic correctness of the input language text

6. Code generator

- o Walks the syntax tree
- o Generates abstract machine code

7. Additional components:

- o Analyze and transform the syntax tree (improve efficiency and remove redundancy from generated machine code)
- o Dictionaries

7. Design and implementation

- Develop and executable software system
- A stage in software engineering process
- Software design and implementation activities are invariably interleaved
- Software design
 - Creative activity
 - Identify software components and their relationships
 - Take implementation issues into account when developing a design
- Software implementation
 - Process of realizing the design as a program
- Important implementation decision
 - Buy or build the application software? (COTS vs. open-source)

Object-oriented design using the UML

- Object-oriented system
 - Made up of interacting objects that maintain their own local state (provide operations on that state)
- Design object classes and the relationships between them
- What to do to develop a system design
 - Understand and define the context and external interactions with the system
 - Design the system architecture
 - Identify the principal objects in the system
 - Develop design models
 - Specify interfaces
- Design is not a clear-cut, sequential process (ideas, propose solutions, refine solutions, ideas etc.)

System context and interactions

- Understand relationships between software and external environment
- Decide how to provide required system functionality, how to structure the system to communicate with its environment
- Set the system boundaries helps to decide what features are implemented
- System context: a structural model that demonstrates other systems in the environment
 - May be represented using associations (relationships)
 - Nature of relationships are specified
 - Using a simple block diagram
- Interaction model: a dynamic model that shows how the system interacts with its environment
 - Model with an abstract approach (not too much detail)
 - Use a case model
 - Cases should be described in structured natural language

Architectural design

- Base knowledge: system context and interactions plus general knowledge of principles of architectural design (with more detailed domain knowledge)
- Identify major components and their interactions
- Organize components using an architectural pattern (layered, client-server etc.; optional)

Object class identification

- Prerequisites:
 - Ideas about the essential objects in the system to be designed
- High-level system objects are needed to encapsulate the system interactions defined in the use cases (usually)
- Methods how to identify object classes
 - Grammatical analysis of a natural language description of the system to be constructed (nouns: objects, attributes; verbs: operations, services)
 - Tangible entities in the application domain (aircraft, roles, events, interactions, locations etc.)
 - Scenario-based analysis where various scenarios of system use are identified and analyzed in turn (team responsible for the analysis must identify the required objects, attributes and operations)
- Use several knowledge sources to discover object classes
- Information from application domain knowledge or scenario analysis to refine/extend initial objects (can be collected from requirements documents, discussions with users, from analyses of existing systems)
- Focus on objects themselves (without thinking about how these might be implemented)
- Look for common features (then design inheritance hierarchy for the system)

Design models

- Show objects or object classes in the system
- Show associations and relationships between objects
- Synonym: system models
- A bridge between the system requirements and the implementation of a system
- They have to be abstract (to focus on relationships)
- They have to include enough detail for programmers (to make implementation decisions)
- Solution: develop models at different levels of detail
- Important step: decide on the needed design models and the level of detail required in these models (depends on type of system)
- With UML, develop 2 kinds of models
 - Structural models: static structure of the system using objects and relationships
 - Dynamic models: dynamic structure of the system; show interactions between system objects

Useful models for adding detail to use case and architectural models:

1. Subsystem models
 - Show logical groupings of objects into coherent subsystems
 - Represented using a form of class diagram (each subsystem shown as a package with enclosed objects)
 - Subsystem models are static models (structural)
2. Sequence models
 - Show sequence of object interactions
 - Represented using a UML sequence / collaboration diagram
 - Sequence models are dynamic models
3. State machine models
 - Show how individual objects change their state in response to events
 - Represented using state diagrams (in the UML)
 - State machine models are dynamic models

Interface specification

- Important part of a design process
- To design objects and subsystem in parallel
- Concerned with specifying the detail of the interface to an object / group of objects
- Define signatures and semantics of the services (provided by the object / group of objects)
- Can be specified in the UML (same notation as a class diagram)
- UML stereotype <<interface>> should be included
- No attribute section
- Semantics of the interface
 - Defined using the object constraint language (OCL)
- Do not include details of the data representing in an interface design (no attributes in the specification)
- Design that is more maintainable
- Not a simple 1:1 relationship between objects and interfaces
 - One object may have several interfaces
 - Supported directly in Java
 - Group of objects may all be accessed through a single interface

Design patterns

- Pattern is
 - A description of the problem and the essence of its solution (solution may be reused in different settings)
 - Not a detailed specification
 - A description of “accumulated wisdom and experience”
 - A well-trying solution to a common problem
- Usually associated with object-oriented design
- Published patterns often rely on object characteristics (e.g. inheritance, polymorphism)
- A way of reusing the knowledge and experience of other designers
- Essential elements of design patterns
 - A name (unique, meaningful)
 - A description of the problem area (when to apply the pattern)
 - A solution description of the parts of the design solution, their relationships and their responsibilities (not a concrete design description; a template for a design solution)
 - A statement of the consequences (results, trade-offs); helps to decide whether or not to use the design pattern
- Any design problem may have an associated pattern that can be applied
- Patterns support high-level concept reuse
- A great idea, but requires experience of software design to use them effectively
 - Recognize situations where patterns can be applied
- Sample: observer pattern

Implementation issues

- Create an executable version of the software
- May involve
 - developing programs in high- or low-level programming languages
 - Tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization

SOFTWARE ENGINEERING

Reuse

- Due to costs and schedule pressure
- Develop new systems more quickly, with fewer development risks and lower costs
- Software should be more reliable

Possible at a number of different levels:

1. Abstraction level

- Don't reuse software directly
- Use knowledge of successful abstractions in the design of your software
- Ways of representing abstract knowledge for reuse: design patterns and architectural patterns

2. Object level

- Directly reuse objects from a library (rather than writing code)
- Find appropriate libraries and discover if objects and methods offer the functionality needed
- Sample: process mail messages (in Java)

3. Component level

- Components: collections of objects and object classes
- Adapt and extend components by adding some source code
- Sample: user interface

4. System level

- Reuse entire application systems
- Involves some kind of configuration of these systems
- Done by adding/modifying code or by using system's own configuration interface
- Used by most commercial systems (generic COTS systems are adapted and reused)

5. Costs associated with reuse:

- Costs of time spent in looking for software to reuse and assessing whether or not it meets the needs
- Costs of buying the reusable software (can be very high for large COTS systems)
- Costs of adapting and configuring the reusable software components or systems to reflect the requirements
- Costs of integrating reusable software elements with each other and with the new code that has been developed

Configuration Management

- Change management is absolutely essential
- General process of managing a changing software system
- Aim: support the system integration process so that all developers can
 - Access the project code and documents in a controlled way
 - Find out what changes have been made
 - Compile and link components to create a system
- Fundamental configuration activities
 - Version management (keep track of different versions; stop overwriting code being submitted by someone else)
 - System integration (define what versions of components are used to create each version of a system; description is used to build a system automatically by compiling and linking the requirements components)
 - Problem tracking (allow users to report bugs and other problems)

Host-target Development

- Most software development is based on host-target model
 - Software is developed on one machine (host)
 - Software runs on separate machines (targets)
- Alternate perspective: development-execution platform
- Often use of simulators in case of development for
 - Embedded systems
 - Different platforms or expensive hard drive
- Simulators
 - Speed up development process for embedded systems
 - Are expensive to develop
 - Usually only available for most popular hardware architectures
- In case middleware is being used, the software is usually transferred to the execution platform for testing (due to license restrictions, etc.)
- Integrated tools of a software development platform (to support software engineering processes)
 - An integrated compiler and syntax-directed editing systems
 - Language debugging system
 - Graphical editing tools (edit UML models)
 - Testing tools (JUnit) to automatically run a set of tests on a new version of a program
 - Project support tools (help to organize code for different development projects)
 - Advanced tools like static analyzers
- IDE: an integrated development environment, made up of several software development tools and that support different aspects of software development
- Document decisions on hardware and software deployment using UML deployment diagrams

Issues to consider when to decide how the developed software will be deployed for distributed systems:

- Hardware and software requirements of a component
 - A component must be deployed on a platform that provides the required hardware/software support (in case the component has been designed for a specific hardware architecture or relies somehow on it)
- Availability requirements of the system
 - High-availability systems may require components to be deployed on more than one platform
 - An alternative implementation of the component is available in case of platform failure
- Component communications
 - Deploy components on the same platform (or those physically close) in case of high level of communications traffic between components
 - Reduces communications latency, delay between the time a message is sent by one component and received by another

Open source Development

- An approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Roots: Free Software Foundation
- Successful open source system still rely on a core group of developers who control changes to the software
- Samples: Linux, Java, Apache, MySQL (IBM, Sun)
- Fairly cheap/free to acquire open source software
- Emerging/common business models: selling support for a product rather than selling a software product

Open source Licensing

- Common licenses: GPL, LGPL, BSD
- Establish a system for maintaining information about open source components that are downloaded and used
- Be aware of the different types of licenses and understand how a component is licensed before it is used
- Be aware of evolution pathways for components
- Educate people about open source
- Have auditing systems in place
- Participate in the open source community
- Trend:
 - Increasingly difficult to build a business by selling specialized software systems
 - Sell support and consultancy to software users for their open source software

8. Component-based Software Engineering

Many new business systems are now developed by configuring off-the-shelf systems. However, when a company cannot use an off-the-shelf system because it does not meet their requirements, the software they need has to be specially developed. For custom software, component-based software engineering is an effective, reuse-oriented way to develop new enterprise systems.

- Component-based software engineering (CBSE) emerged in the late 1990s as an approach to software systems development based on reusing software components.
- Its creation was motivated by object-oriented development by designers' to have extensive reuse. The developers need to have access to the component source code.
- This meant that selling or distributing objects as individual reusable components was practically impossible.
- Components are higher-level abstractions than objects and are defined by their interfaces.
- They are usually larger than individual objects and all implementation details are hidden from other components.
- CBSE is the process of defining, implementing, and integrating or composing loosely coupled independent components into systems.

The essentials of component-based software engineering are:

- Independent components that are completely specified by their interfaces.
- There should be a clear separation between the component interface and its implementation.
- Component standards that facilitate the integration of components are embodied in a component model.
- They define, at the very minimum, how component interfaces should be specified and how components communicate.
- Some models go much further and define interfaces that should be implemented by all conformant components.
- If components conform to standards, then their operation is independent of their programming language.
- Components written in different languages can be integrated into the same system.
- Middleware that provides software support for component integration.
- To make independent, distributed components work together, you need middleware support that handles component communications.
- Middleware for component support handles low-level issues efficiently and allows you to focus on application-related problems. In addition, middleware for component support may provide support for resource allocation, transaction management, security, and concurrency.
- A development process that is geared to component-based software engineering.
- You need a development process that allows requirements to evolve, depending on the functionality of available components

SOFTWARE ENGINEERING

Components and Component Models

Councill and Heineman (2001) Definition of a Component:

"A software element that conforms to a standard component model and can be independently deployed and composed without modification according to a composition standard."

This definition is essentially based on standards so that a software unit that conforms to these standards is a component.

Essential characteristics of a component as used in CBSE.

Component Characteristic	Description
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of a component. Rather, it is deployed by the service provider.
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.

SOFTWARE ENGINEERING

Szyperski (2002) Definition of a Component: does not mention standards in his definition of a component but focuses instead on the key characteristics of components:

"A software component is a unit of composition with contractually-specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

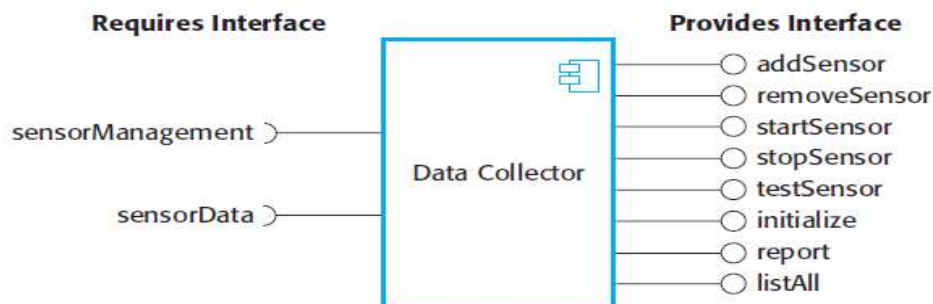
A useful way of thinking about a component is as a provider of one or more services. When a system needs a service, it calls on a component to provide that service without caring about where that component is executing or the programming language used to develop the component.

Viewing a component as a service provider emphasizes two critical characteristics of a reusable component:

- The component is an independent executable entity that is defined by its interfaces.
- No need to have any knowledge of its source code and how to use it.
- It can either be referenced as an external service or included directly in a program.
- The services offered by a component are made available through an interface and all interactions are through that interface.
- The component interface is expressed in terms of parameterized operations
- The internal state of the component is never exposed.
- Components have two related interfaces. These interfaces reflect the services that the component provides and the services that the component requires to operate correctly:



- The 'provides' interface defines the services provided by the component.
- This interface, essentially, is the component API. It defines the methods that can be called by a user of the component.
- In a UML component diagram, the 'provides' interface for a component is indicated by a circle at the end of a line from the component icon.
- The 'requires' interface specifies what services must be provided by other components in the system if a component is to operate correctly.
- If these are not available, then the component will not work.
- This does not compromise the independence or deployability of a component because the 'requires' interface does not define how these services should be provided.
- In the UML, the symbol for a 'requires' interface is a semicircle at the end of a line from the component icon. Notice that 'provides' and 'requires' interface icons can fit together like a ball and socket.



9. Distributed Software Engineering

- Virtually all large computer-based systems are now distributed systems.
- A distributed system is one involving several computers.
- It is in contrast with centralized systems where all of the system components execute on a single computer.

Definition: “Distributed systems is a collection of independent computers that appears to the user as a single coherent system.” - Tanenbaum and Van Steen (2007)

Coulouris (2005) identified the following advantages of using a distributed approach to systems development:

1. **Resource sharing :** A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.
 2. **Openness:** Distributed systems are normally open systems, which means that they are designed around standard protocols that allow equipment and software from different vendors to be combined.
 3. **Concurrency:** In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.
 4. **Scalability:** In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.
 5. **Fault tolerance:** The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures (see Chapter 13). In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.
- Distributed systems are inherently more complex than centralized systems.
 - They are difficult to design, implement, and test.
 - It is harder to understand the properties of distributed systems because of the complexity of the interactions between system components and the system infrastructure.
 - distributed systems are unpredictable in their response.
 - The response time depends on the overall load on the system, its architecture and the network load.
 - The most important development that has affected distributed software systems in the past few years is the service-oriented approach.

Distributed Systems Issues

- Distributed systems are more complex than systems that run on a single processor.
- It is practically impossible to have a top-down model for control of these systems
- The functionality nodes in the system are often independent systems with no single authority in charge of them.
- The network connecting these nodes is a separately managed system.
- It is a complex system in its own right and cannot be controlled by the owners of systems using the network.

SOFTWARE ENGINEERING

- There is therefore an inherent unpredictability in the operation of distributed systems that has to be taken into account by the system designer.

Some of the most important design issues that have to be considered in distributed systems engineering are:

1. **Transparency:** To what extent should the distributed system appear to the user as a single system? When it is useful for users to understand that the system is distributed?
2. **Openness:** Should a system be designed using standard protocols that support interoperability or should more specialized protocols be used that restrict the freedom of the designer?
3. **Scalability:** How can the system be constructed so that it is scalable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?
4. **Security:** How can usable security policies be defined and implemented that apply across a set of independently managed systems?
5. **Quality of Service:** How should the quality of service that is delivered to system users be specified and how should the system be implemented to deliver an acceptable quality of service to all users?
6. **Failure management:** How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?

In an ideal world, the fact that a system is distributed would be transparent to users. This means that users would see the system as a single system whose behavior is not affected by the way that the system is distributed. Practically, this is not possible to achieve.

The reasons being -

- Central control of a distributed system is impossible and, as a result, individual computers in a system may behave differently at different times.
- It always takes a finite length of time for signals to travel across a network, network delays are unavoidable.
- The length of these delays depends on the location of resources in the system, the quality of the user's network connection, and the network load.
- To achieve transparency of the system we need to create abstractions of the resources in a distributed system. This helps us to modify the resources without having to make changes in the application system.
- Middleware as it is called is used to map the logical resources referenced by a program onto the actual physical resources, and to manage the interactions between these resources.
- It is best to expose the distribution of the physical system to users.
- They can then be prepared for some of the consequences of distribution such as network delays, remote node failures, etc.
- Open distributed systems are systems that are built according to generally accepted standards.
- This means that components from any supplier can be integrated into the system and can interoperate with the other system components.
- At the networking level, openness is now taken for granted with systems conforming to Internet protocols but at the component level, openness is still not universal.

SOFTWARE ENGINEERING

The CORBA standard (Pope, 1997) developed in the 1990s was intended to achieve this but this never achieved a critical mass of adopters. Rather, many companies chose to develop systems using proprietary standards for components from companies such as Sun and Microsoft. Web service standards for service-oriented architectures were developed to be open standards. However, there is significant resistance to these standards because of their perceived inefficiency. The scalability of a system reflects its ability to deliver a high quality of service as demands on the system increase. Neuman (1994) identifies three dimensions of scalability:

1. Size It should be possible to add more resources to a system to cope with increasing numbers of users.
2. Distribution It should be possible to geographically disperse the components of a system without degrading its performance.
3. Manageability It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations.

However, when a system is distributed, the number of ways that the system may be attacked is significantly increased, compared to centralized systems

1. The types of attacks that a distributed system must defend itself against are the following:
2. Interception, where communications between parts of the system are intercepted by an attacker so that there is a loss of confidentiality.
3. Interruption, where system services are attacked and cannot be delivered as expected. Denial of service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
4. Modification, where data or services in the system are changed by an attacker.
5. Fabrication, where an attacker generates information that should not exist and then uses this to gain some privileges. For example, an attacker may generate a false password entry and use this to gain access to a system.

The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that is acceptable to its users. Ideally, the QoS requirements should be specified in advance and the system designed and configured to deliver that QoS. Unfortunately, this is not always practicable, for the reasons:

1. It may not be cost effective to design and configure the system to deliver a high QoS under peak load.
2. This could involve making resources available that are unused for much of the time.
3. One of the main arguments for 'cloud computing' is that it partially addresses this problem.
4. Using a cloud, it is easy to add resources as demand increases.
5. The QoS parameters may be mutually contradictory. For example, increased reliability may mean reduced throughput, as checking procedures are introduced to ensure that all system inputs are valid.

QoS is particularly critical when the system is dealing with time-critical data such as sound or video streams. In these circumstances, if the QoS falls below a threshold value then the sound or video may become so degraded that it is impossible to understand. Systems dealing with sound and video should include QoS negotiation and management components. These should evaluate the QoS requirements against the available resources and, if these are insufficient, negotiate for more resources or for a reduced QoS target.

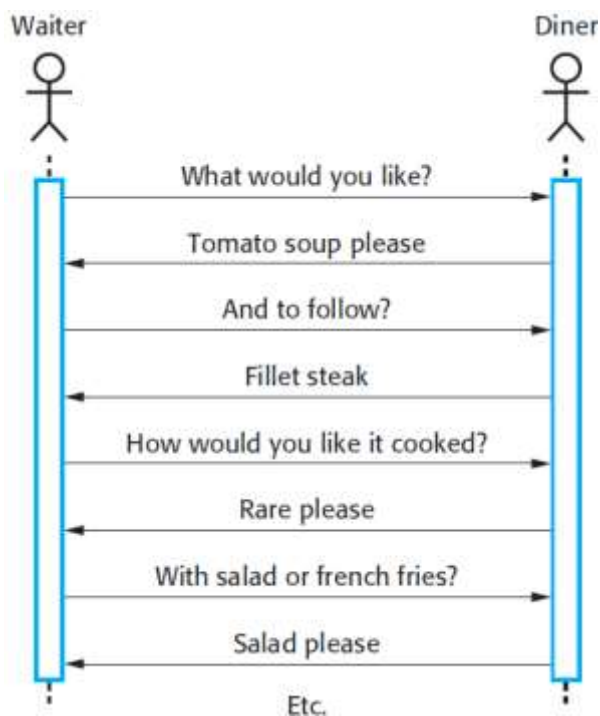
Failure management involves applying the fault tolerance techniques. Distributed systems should therefore include mechanisms for discovering if a component of the system has failed, should continue to deliver as many services as possible in spite of that failure and, as far as possible, should automatically recover from the failure.

Models of Interaction

There are two fundamental types of interaction that may take place between the computers in a distributed computing system:

1. Procedural Interaction: involves one computer calling on a known service offered by some other computer and (usually) waiting for that service to be delivered.
2. Message-based Interaction: Message-based interaction involves the 'sending' computer defining information about what is required in a message, which is then sent to another computer.

PROCEDURAL INTERACTION



- Procedural communication in a distributed system is usually implemented using remote procedure calls (RPCs).
- In RPC one component calls another component as if it was a local procedure or method.
- The middleware in the system intercepts this call and passes it to a remote component.
- This carries out the required computation and, via the middleware, returns the result to the calling component.
- In Java, remote method invocations (RMI) are comparable with, though not identical to, RPCs.
- The RMI framework handles the invocation of remote methods in a Java program.
- RPCs require a 'stub' for the called procedure to be accessible on the computer that is initiating the call.
- The stub is called and it translates the procedure parameters into a standard representation for transmission to the remote procedure.
- Through the middleware, it then sends the request for execution to the remote procedure.
- The remote procedure uses library functions to convert the parameters into the required format, carries out the computation, and then communicates the results via the 'stub' that is representing the caller.

SOFTWARE ENGINEERING

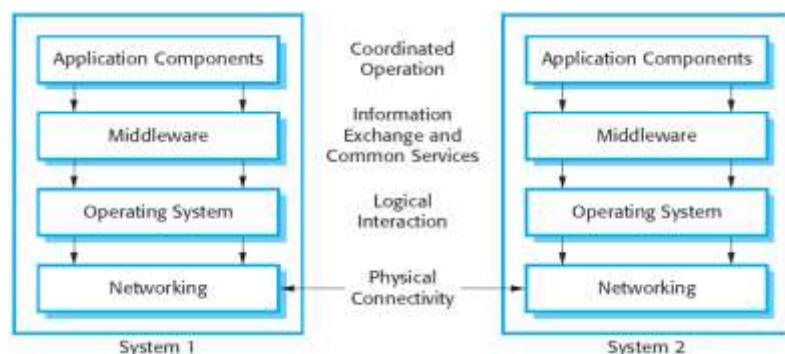
MESSAGE-BASED INTERACTION

- Normally involves one component creating a message that details the services required from another component.
- Through the system middleware, this is sent to the receiving component.
- The receiver parses the message, carries out the computations, and creates a message for the sending component with the required results.
- This is then passed to the middleware for transmission to the sending component.

A problem with the RPC approach to interaction is that both the caller and the requestor need to be available at the time of the communication, and they must know how to refer to each other.

Middleware

- The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor.
- Models of data, information representation, and protocols for communication may all be different.
- A distributed system therefore requires software that can manage these diverse parts, and ensure that they can communicate and exchange data.
- The term 'middleware' is used to refer to this software—it sits in the middle between the distributed components of the system.



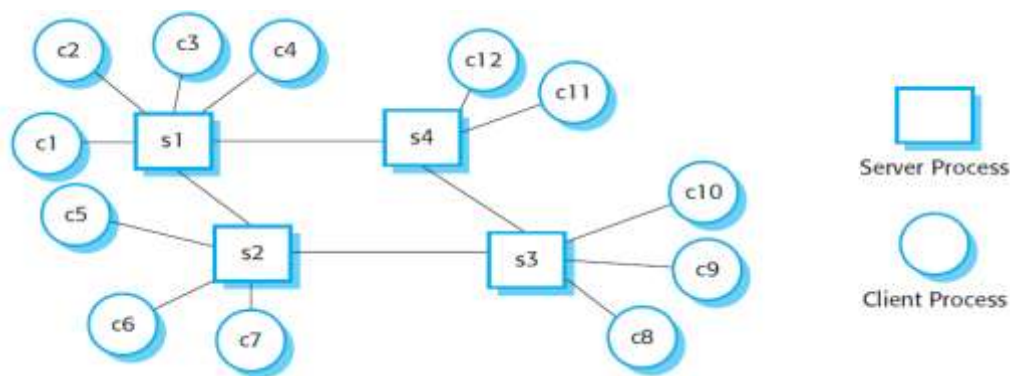
In a distributed system, middleware normally provides two distinct types of support:

1. **Interaction support:** where the middleware coordinates interactions between different components in the system provides location transparency and supports parameter conversion if different programming languages are used to implement components, event detection, and communication, etc.
2. **Provision of common services:** where the middleware provides reusable implementations of services that may be required by several components in the distributed system. By using these common services, components can easily interoperate and provide user services in a consistent way.

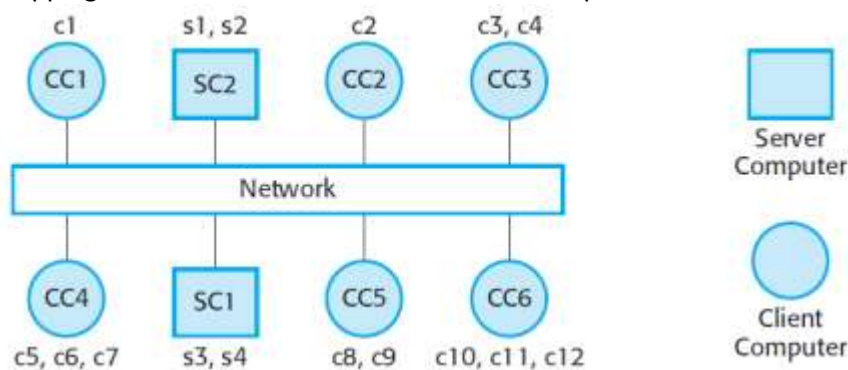
SOFTWARE ENGINEERING

Client-server computing

- Distributed systems that are accessed over the Internet are normally organized as client-server systems.
- In a client-server system, the user interacts with a program running on their local computer.
- This interacts with another program running on a remote computer (e.g., a web server).
- The remote computer provides services, such as access to web pages, which are available to external clients. This client-server model is a very general architectural model of an application.
- It can be used as a logical interaction model where the client and the server run on the same computer.
- In client-server architecture, an application is modeled as a set of services that are provided by servers.
- Clients may access these services and present results to end users.
- Clients need to be aware of the servers that are available but do not know of the existence of other clients. Clients and servers are separate processes.



Mapping of clients and servers to networked computers:



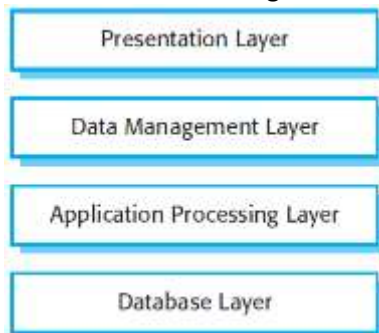
Client-Server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information. Consequently, the design of the architecture of distributed client-server systems should be structured into several logical layers, with clear interfaces between these layers. A Client-Server application is structured into four layers:

1. A presentation layer that is concerned with presenting information to the user and managing all user interaction;
2. A data management layer that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, etc.;
3. An application processing layer that is concerned with implementing the logic of the application and so providing the required functionality to end users;

SOFTWARE ENGINEERING

4. A database layer that stores the data and provides transaction management services, etc.

Client–Server architectures distribute logical layers in different ways. The client–server model also underlies the notion of Software As A Service (SaaS), an increasingly important way of deploying software and accessing it over the Internet.



Architectural patterns for distributed systems

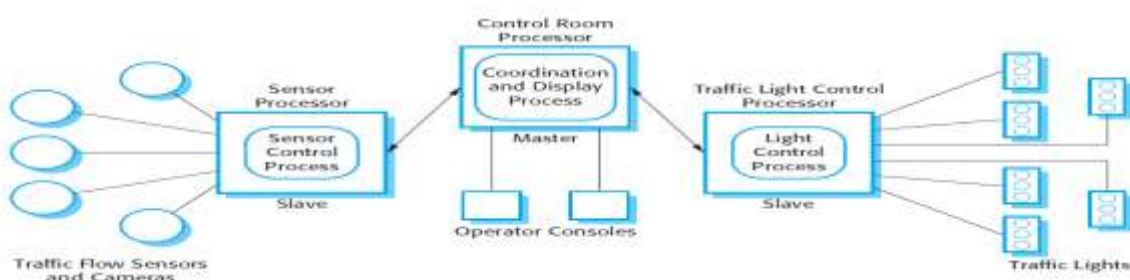
Designers of distributed systems have to organize their system designs to find a balance between performance, dependability, security, and manageability of the system. When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system.

The 5 architectural styles are:

1. **Master-slave architecture:** which is used in real-time systems in which guaranteed interaction response times are required.
2. **Two-tier client–server architecture:** which is used for simple client–server systems, and in situations where it is important to centralize the system for security reasons. In such cases, communication between the client and server is normally encrypted.
3. **Multitier client–server architecture:** which is used when there is a high volume of transactions to be processed by the server.
4. **Distributed component architecture:** which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client–server systems.
5. **Peer-to-peer architecture:** which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

Master-slave architectures

- Master-slave architectures are commonly used in real-time systems where there may be separate processors associated with data acquisition from the system’s environment, data processing and computation and actuator management.
- The ‘master’ process is usually responsible for computation, coordination and communications and it controls the ‘slave’ processes.
- ‘Slave’ processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.



SOFTWARE ENGINEERING

1. The Traffic Controller Model illustrates this architectural model.
2. It has three logical processes that run on separate processors.
3. The master process is the control room process, which communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.
4. A set of distributed sensors collects information on the traffic flow.
5. The sensor control process polls the sensors periodically to capture the traffic flow information and collates this information for further processing.
6. The sensor processor is itself polled periodically for information by the master process that is concerned with displaying traffic status to operators, computing traffic light sequences and accepting operator commands to modify these sequences.
7. The control room system sends commands to a traffic light control process that converts these into signals to control the traffic light hardware.
8. The master control room system is itself organized as a client-server system, with the client processes running on the operator's consoles.

Two-tier client-server architectures

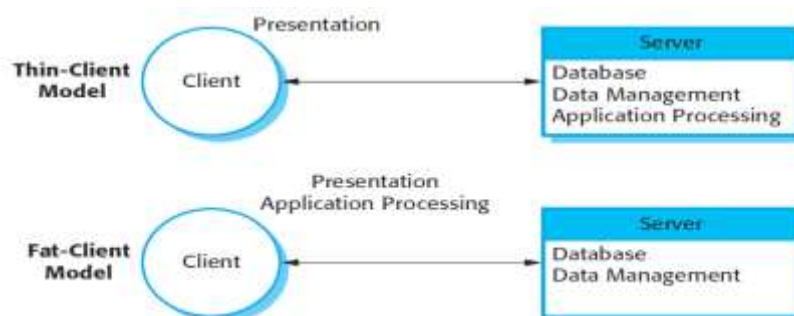
- In a two-tier client-server architecture, the system is implemented as a single logical server plus an indefinite number of clients that use that server.
 1. Thin-client model, where the presentation layer is implemented on the client and all other layers (data management, application processing and database) are implemented on a server.
 2. Fat-client model, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server.

1. Thin client model

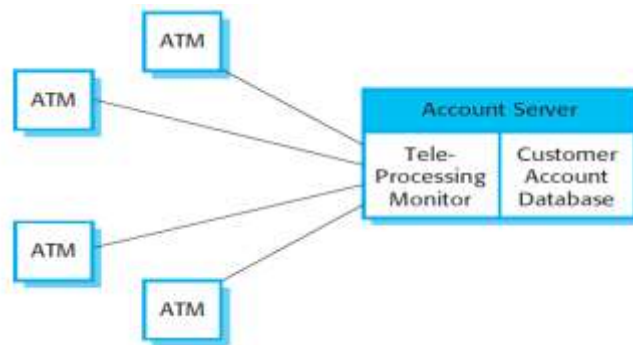
- Used when legacy systems are migrated to client server architectures.
- The legacy system acts as a server in its own right with a graphical interface implemented on a client.
- A major disadvantage is that it places a heavy processing load on both the server and the network.

2. Fat client model

- More processing is delegated to the client as the application processing is locally executed.
- Most suitable for new C/S systems where the capabilities of the client system are known in advance.
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.



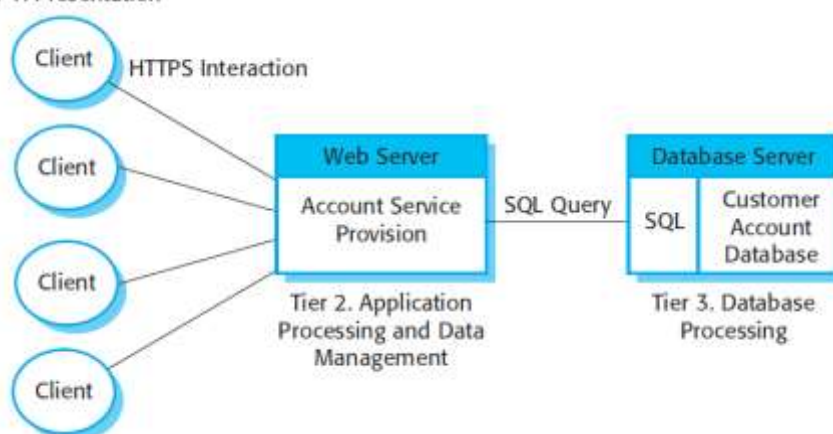
1000



1. The ATM is the client computer and the server is, typically, a mainframe running the customer account database.
2. A mainframe computer is a powerful machine that is designed for transaction processing.
3. It can therefore handle the large volume of transactions generated by ATMs, other teller systems, and online banking.
4. The software in the teller machine carries out a lot of the customer-related processing associated with a transaction.
5. The ATMs do not connect directly to the customer database, but rather to a teleprocessing monitor.
6. A teleprocessing (TP) monitor is a middleware system that organizes communications with remote clients and serializes client transactions for processing by the database.
7. This ensures that transactions are independent and do not interfere with one other.
8. Using serial transactions means that the system can recover from faults without corrupting the system data.

- In a 'multi-tier client-server' architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.
- This avoids problems with scalability and performance if a thin-client two-tier model is chosen, or problems of system management if a fat-client model is used.

Tier 1. Presentation



SOFTWARE ENGINEERING

1. An Internet banking system is an example of a multi-tier client–server architecture, where there are three tiers in the system.
2. The bank’s customer database (usually hosted on a mainframe computer) provides database services.
3. A web server provides data management services such as web page generation and some application services.
4. Application services such as facilities to transfer cash, generate statements, pay bills, and so on are implemented in the web server and as scripts that are executed by the client.
5. The user’s own computer with an Internet browser is the client.
6. This system is scalable because it is relatively easy to add servers (scale out) as the number of customers increase.
7. In this case, the use of a three-tier architecture allows the information transfer between the web server and the database server to be optimized.
8. The communications between these systems can use fast, low-level data exchange protocols.
9. Efficient middleware that supports database queries in SQL (Structured Query Language) is used to handle information retrieval from the database.
10. The three-tier client–server model can be extended to a multi-tier variant, where additional servers are added to the system.

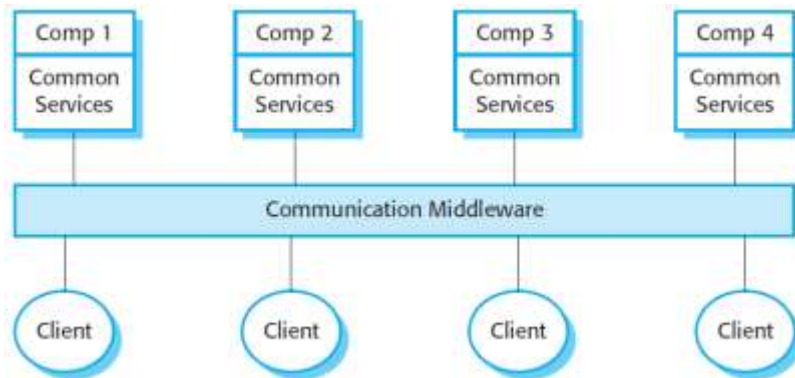
Use of Client-Server architectural patterns:

Architecture	Applications
Two-tier client–server architecture with thin clients	Legacy system applications that are used when separating application processing and data management is impractical. Clients may access these as services, as discussed in Section 18.4. Computationally intensive applications such as compilers with little or no data management. Data-intensive applications (browsing and querying) with non-intensive application processing. Browsing the Web is the most common example of a situation where this architecture is used.
Two-tier client-server architecture with fat clients	Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client. Applications where computationally intensive processing of data (e.g., data visualization) is required. Mobile applications where internet connectivity cannot be guaranteed. Some local processing using cached information from the database is therefore possible.
Multi-tier client–server architecture	Large-scale applications with hundreds or thousands of clients. Applications where both the data and the application are volatile. Applications where data from multiple sources are integrated.

SOFTWARE ENGINEERING

Distributed component architectures

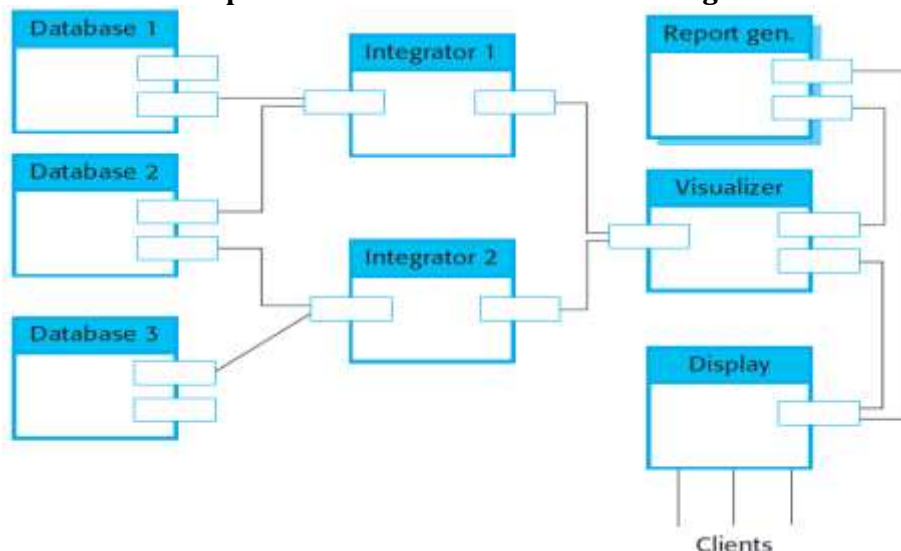
- There is no distinction in a distributed component architectures between clients and servers.
- Each distributable entity is an object that provides services to other components and receives services from other components.
- Component communication is through a middleware system.
- However, distributed component architectures are more complex to design than C/S systems.



The benefits of using a distributed component model for implementing distributed systems are the following:

1. It allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network. There is no need to decide in advance whether a service is part of a data management layer, an application layer, etc.
2. It is a very open system architecture that allows new resources to be added as required. New system services can be added easily without major disruption to the existing system.
3. The system is flexible and scalable. New components or replicated components can be added as the load on the system increases, without disrupting other parts of the system.
4. It is possible to reconfigure the system dynamically with components migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A service-providing component can migrate to the same processor as service-requesting objects, thus improving the performance of the system.

Distributed Component Architecture for Data-Mining:



SOFTWARE ENGINEERING

1. Data mining systems are a good example of a type of system in which a distributed component architecture is the best architectural pattern to use.
2. A data mining system looks for relationships between the data that is stored in a number of databases
3. Each sales database can be encapsulated as a distributed component with an interface that provides read-only access to its data.
4. Integrator components are each concerned with specific types of relationships
5. They collect information from all of the databases to try to deduce the relationships.
6. There might be an integrator component that is concerned with seasonal variations in goods sold, and another that is concerned with relationships between different types of goods.
7. Visualizer components interact with integrator components to produce a visualization or a report on the relationships that have been discovered.
8. Because of the large volumes of data that are handled, visualizer components normally present their results graphically.
9. Finally, a display component may be responsible for delivering the graphical models to clients for final presentation

Distributed component architectures suffer from two major disadvantages:

1. They are more complex to design than client–server systems. Multi-layer client–server systems appear to be a fairly intuitive way to think about systems. They reflect many human transactions where people request and receive services from other people who specialize in providing these services. By contrast, distributed component architectures are more difficult for people to visualize and understand.
2. Standardized middleware for distributed component systems has never been accepted by the community. Rather different vendors, such as Microsoft and Sun, have developed different, incompatible middleware. This middleware is complex and reliance on it increases the overall complexity of distributed component systems.

Peer-to-peer architectures

- Peer to peer (p2p) systems are decentralised systems where computations may be carried out by any node in the network.
- The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.
- Most p2p systems have been personal systems but there is increasing business use of this technology.

P2P architectural models

- The logical network architecture
 - Decentralized architectures;
 - Semi-centralized architectures.
- Application architecture
 - The generic organization of components making up a p2p application.
 - Focus here on network architectures.

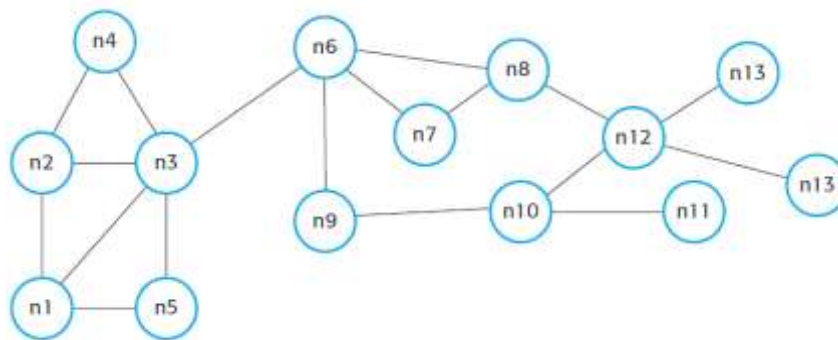
SOFTWARE ENGINEERING

Peer-to-peer technologies have mostly been used for personal rather than business systems. For example, file-sharing systems based on the Gnutella and BitTorrent protocols are used to exchange files on users' PCs. Instant messaging systems such as ICQ and Jabber provide direct communications between users without an intermediate server. SETI@home is a long-running project to process data from radio telescopes on home PCs to search for indications of extra-terrestrial life. Voice over IP (VOIP) phone services, such as Skype; rely on peer-to-peer communication between the parties involved in the phone call or conference. However, peer-to-peer systems are also being used by businesses to harness the power in their PC networks. Intel and Boeing have both implemented p2p systems for computationally intensive applications.

It is appropriate to use a peer-to-peer architectural model for a system in two circumstances:

1. Where the system is computationally intensive and it is possible to separate the processing required into a large number of independent computations. For example, a peer-to-peer system that supports computational drug discovery distributes computations that look for potential cancer treatments by analysing a huge number of molecules to see if they have the characteristics required to suppress the growth of cancers. Each molecule can be considered separately so there is no need for the peers in the system to communicate.
2. Where the system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally stored or managed. Examples of such applications include file-sharing systems that allow peers to exchange local files such as music and video files, and phone systems that support voice and video communications between computers.

A Decentralized P2P Architecture:



If someone needs a document that is stored somewhere on the network, they issue a search command, which is sent to nodes in their 'locality'. These nodes check whether they have the document and, if so, return it to the requestor. If they do not have it, they route the search to other nodes.

This decentralized architecture has advantages in that it is highly redundant and hence both fault-tolerant and tolerant of nodes disconnecting from the network. However, the disadvantages here are that many different nodes may process the same search, and there is also significant overhead in replicated peer communications.

SOFTWARE ENGINEERING

A Semi-Centralized Architecture:



Instant Messaging Service: Network nodes communicate with the server (indicated by dashed lines) to find out what other nodes are available. Once these nodes are discovered, direct communications can be established and the connection to the server is unnecessary

Software as a Service

Software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet.

- Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
- The software is owned and managed by a software provider, rather than the organizations using the software.
- Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription.

Key elements of SaaS:

- Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
- The software is owned and managed by a software provider, rather than the organizations using the software.
- Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Sometimes, the software is free for anyone to use but users must then agree to accept advertisements, which fund the software service.

SaaS and SOA

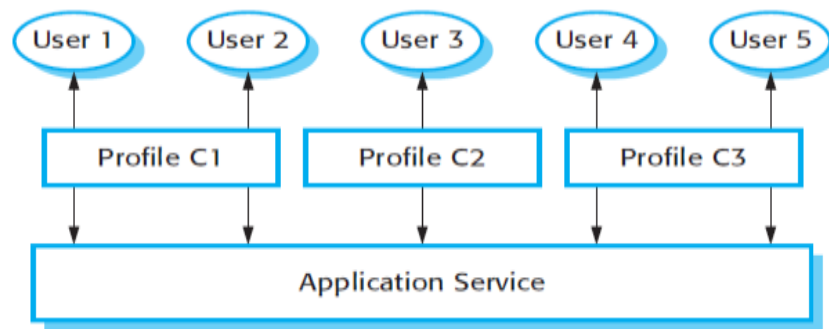
- Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user's data and state during an interaction session. Transactions are usually long transactions e.g. editing a document.
- Service-oriented architecture (SOA) is an approach to structuring a software system as a set of separate, stateless services. These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something then returns a result.

Implementation factors for SaaS

- **Configurability:** How do you configure the software for the specific requirements of each organization?
- **Multi-tenancy:** How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
- **Scalability:** How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

SOFTWARE ENGINEERING

Configuration of a software system offered as a service:



Service configuration:

- Branding, where users from each organization, are presented with an interface that reflects their own organization.
- Business rules and workflows, where each organization defines its own rules that govern the use of the service and its data.
- Database extensions, where each organization defines how the generic service data model is extended to meet its specific needs.
- Access control, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

Multi-tenancy:

- Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources.
- It must appear to each user that they have the sole use of the system.
- Multi-tenancy involves designing the system so that there is an absolute separation between the system functionality and the system data.

A multi-tenant database:

Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

- This diagram shows four users of the application service, who work for three different customers of the service provider.
- Users interact with the service through a customer profile that defines the service configuration for their employer.
- Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources.
- It must appear to each user that they have the sole use of the system.
- Multi-tenancy involves designing the system so that there is an absolute separation between the system functionality and the system data.
- Design the system so that all operations are stateless.
- Data should either be provided by the client or should be available in a storage system or database that can be accessed from any system instance.
- Relational databases are not ideal for providing multi-tenancy.

SOFTWARE ENGINEERING

Scalability:

- Develop applications where each component is implemented as a simple stateless service that may be run on any server.
- Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request).
- Manage resources, such as network and database connections, as a pool so that no single server is likely to run out of resources.
- Design your database to allow fine-grain locking. That is, do not lock out whole records in the database when only part of a record is in use.

10. Planning a Software Project

Process Planning

Planning is the most important project management activity. It has two basic objectives—

1. Establish reasonable cost & schedule
2. Quality goals for the project & to draw out a plan to deliver the project goals.

Planning Guidelines

- A project succeeds if it meets its cost, schedule, and quality goals.
- Without the project goals being defined, it is not possible to even declare if a project has succeeded.
- Without detailed planning, no real monitoring or controlling of the project is possible.
- Projects are rushed toward implementation with not enough effort spent on planning. No amount of technical effort later can compensate for lack of careful planning.
- Lack of proper planning is a sure ticket to failure for a large software project.
- The inputs to the planning activity are the requirements specification and maybe the architecture description.
- A very detailed requirements document is not essential for planning, but for a good plan all the important requirements must be known, and it is highly desirable that key architecture decisions have been taken.
- Have people familiar with the tasks make the estimate.
- Use several people to make estimates.
- Base estimates on normal conditions, efficient methods, and a normal level of resources.
- Use consistent time units in estimating task times.
- Treat each task as independent, don't aggregate.
- Don't make allowances for contingencies.
- Adding a risk assessment helps avoid surprises to stakeholders.

There are generally two main outputs of the planning activity:

1. the overall project management plan document that establishes
 - the project goals on the cost
 - schedule
 - quality fronts
 - defines the plans for managing risk
 - monitoring the project
2. the detailed plan
 - the detailed project schedule
 - specifying the tasks that need to be performed to meet the goals
 - The resources who will perform them, and their schedule.

The overall plan guides the development of the detailed plan, which then becomes the main guiding document during project execution for project monitoring.

Effort Estimation

For a software development project, overall effort and schedule estimates are essential prerequisites for planning the project. These estimates are needed before development is initiated, as they establish the cost and schedule goals of the project. Without these, even simple questions like “is the project late?” “Are there cost overruns?” and “when is the project likely to complete?” cannot be answered. A more practical use of these estimates is in bidding for software projects, where cost and schedule estimates must be given to a potential client for the development contract. Effort and schedule estimates are also required for determining the staffing level for a project during different phases, for the detailed plan, and for project monitoring. The accuracy with which effort can be estimated clearly depends on the level of information available about the project. The more detailed the information, the more accurate the estimation can be. Of course, even with all the information available, the accuracy of the estimates will depend on the effectiveness and accuracy of the estimation procedures or models employed and the process. If from the requirements specifications, the estimation approach can produce estimates that are within 20% of the actual effort about two-thirds of the time, then the approach can be considered good. Here we discuss two commonly used approaches.

Top-Down Estimation Approach

The top-down method is also known as the analogous method. It is used to determine order of magnitude estimates in the initiation phase of the project. The method uses the actual durations, effort or costs from previous projects as a basis for estimating the effort or costs for the current project.

1. Identify a previous project or section of a previous project that is similar to the current project.
2. Assess the extent to which the current project is similar to the previous project – the comparison factor (e.g 1.5 if the current project is estimated to be 50% larger).
3. Compute the estimate for the current project based on the actual durations, effort or costs from the previous project and the comparison factor

A more general function for determining effort from size that is commonly used is of the form:

$$EFFORT = a * SIZE^b$$

where a and b are constants, and project size is generally in KLOC (Thousand lines of code is treated as KLOC. This metric helps in knowing the size and complexity of the Software Application). Values for these constants for an organization are determined through regression analysis, which is applied to data about the projects that have been performed in the past.

To develop an estimation model we have to follow these steps:

1. Determine a list of potential / most important effort cost drivers.
2. Determine a scaling model for each effort and cost driver.
3. Select initial estimation model.
4. Measure and estimate projects and compare.
5. Evaluate quality of estimation as part of project post-mortem.
6. Update and validate model at appropriate intervals.

SOFTWARE ENGINEERING

The COCOMO Model

The Constructive Cost Model (COCOMO) is an algorithmic software cost estimation model developed by Barry W. Boehm. The model uses a basic regression formula with parameters that are derived from historical project data and current as well as future project characteristics. In the COCOMO model, after determining the initial estimate, some other factors are incorporated for obtaining the final estimate. To do this, COCOMO uses a set of 15 different attributes of a project called cost driver attributes. Examples of the attributes are required software reliability, product complexity, analyst capability, application experience, use of modern tools, and required development schedule. Each cost driver has a rating scale, and for each rating, a multiplying factor is provided.

Boehm proposed three levels of the model: basic, intermediate, detailed.

1. The basic COCOMO'81 model is a single-valued, static model that computes software development effort (and cost) as a function of program size expressed in estimated thousand delivered source instructions (KDSI).
2. The intermediate COCOMO'81 model computes software development effort as a function of program size and a set of fifteen "cost drivers" that include subjective assessments of product, hardware, personnel, and project attributes.
3. The advanced or detailed COCOMO'81 model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process

COCOMO models depends on the two main equations

1. *development effort* : $MM = a * KDSI^b$

based on MM - man-month / person month / staff-month is one month of effort by one person. In COCOMO'81, there are 152 hours per Person month. According to organization this values may differ from the standard by 10% to 20%.

2. *effort and development time (TDEV)* : $TDEV = 2.5 * MM^c$

The coefficients a, b and c depend on the mode of the development. There are three modes of development:

Development Mode	Project Characteristics			
	Size	Innovation	Deadline/constraints	Dev. Environment
Organic	Small	Little	Not tight	Stable
Semi-detached	Medium	Medium	Medium	Medium
Embedded	Large	Greater	Tight	Complex hardware/ customer interfaces

Table 1: development modes

11. Project scheduling and staffing

1. After establishing the effort required, we need to establish the delivery schedule.
2. With the effort estimate (in person-months) it feels easy to estimate the project duration based on convenience & then fix a suitable team size to ensure that the total effort matches the estimate. However person and months are not fully interchangeable in a software project.
3. In software projects—there are dependencies between tasks and they cannot run in parallel (e.g., testing can only be done after coding is done), and a person performing some task in a project needs to communicate with others performing other tasks.
4. For a project with some estimated effort, multiple schedules (or project duration) are indeed possible. For example,
5. If a project effort estimate is 56 person-months
 - a. A total schedule of 8 months is possible with 7 people.
 - b. Alternatively a schedule of 7 months with 8 people is also possible, as is a schedule of approximately 9 months with 6 people.
 - c. But a schedule of 1 month with 56 people is not possible.
 - d. Similarly, no one would execute the project in 28 months with 2 people.
 - e. In other words, once the effort is fixed, there is some flexibility in setting the schedule by appropriately staffing the project, but this flexibility is not unlimited.

Empirical data also suggests that no simple equation between effort and schedule fits well.

1. Fix a reasonable schedule that can be achieved
2. Check if suitable numbers of resources are assigned.
3. Determine the overall schedule as a function of effort. This can be determined from data from completed projects using statistical techniques.
4. The IBM Federal Systems Division found that the total duration, M , in calendar months and E is the effort can be estimated by

$$M = 4.1E^{.36}$$

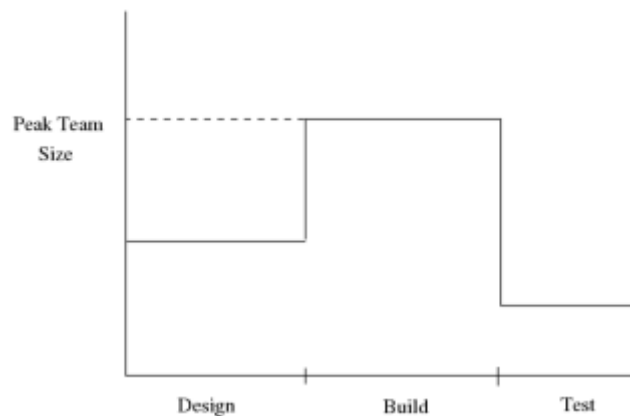
5. In COCOMO, the equation for schedule for an organic type of software is $M = 2.5E^{.38}$

Another method for checking a schedule for medium-sized projects is the rule of thumb called the square root check:

- This check suggests that the proposed schedule can be around the square root of the total effort in person-months.
- This schedule can be met if suitable resources are assigned to the project. Example, if the effort estimate is 50 person-months, a schedule of about 7 to 8 months will be suitable.
- From this macro estimate of schedule, we can determine the schedule for the major milestones in the project.
- To determine the milestones, we must first understand the manpower ramp-up that usually takes place in a project.
- The number of people that can be gainfully utilized in a software project tends to follow the Rayleigh curve.
- That is, in the beginning and the end, few people are needed on the project; the peak team size (PTS) is needed somewhere near the middle of the project; and again fewer people are needed
- after that.

SOFTWARE ENGINEERING

- This occurs because only a few people are needed and can be used in the initial phases of requirements analysis and design. The human resources requirement peaks during coding and unit testing, and during system testing and integration, again fewer people are required.
- Often, the staffing level is not changed continuously in a project and approximations of the Rayleigh curve are used:
 - a) assigning a few people at the start
 - b) having the peak team during the coding phase
 - c) then leaving a few people for integration and system testing.
- For ease of scheduling, particularly for smaller projects, often the required people are assigned together around the start of the project.
 - a) This approach can lead to some people being unoccupied at the start and toward the end.
 - b) This slack time is often used for supporting project activities like training and documentation.
- Given the effort estimate for a phase, we can determine the duration of the phase if we know the manpower ramp-up.
- For these three major phases, the percentage of the schedule consumed in the build phase is smaller than the percentage of the effort consumed because this phase involves more people.
- Similarly, the percentage of the schedule consumed in the design and testing phases exceeds their effort percentages.
- The exact schedule depends on the planned manpower ramp-up, and how many resources can be used effectively in a phase on that project.
- Generally speaking, design requires about a quarter of the schedule, build consumes about half, and integration and system testing consume the remaining quarter. COCOMO gives 19% for design, 62% for programming, and 18% for integration.



Software Configuration Management Plan

The output of the software process is information that may be divided into three broad categories:

1. Computer programs (both source level and executable forms)
2. Documents that describe the computer programs (targeted at both technical practitioners and users)
3. Data (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a software configuration.

SOFTWARE ENGINEERING

- As the software process progresses, the number of Software Configuration Items (SCIs) grows rapidly.
- A System Specification spawns a Software Project Plan and Software Requirements Specification (as well as hardware related documents).
- These in turn spawn other documents to create a hierarchy of information.
- If each SCI simply spawned other SCIs, little confusion would result.
- Unfortunately, another variable enters the process—change.
- Change may occur at any time, for any reason.

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

1. New business or market conditions dictate changes in product requirements or business rules.
2. New customer needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
3. Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
4. Budgetary or scheduling constraints cause a redefinition of the system or product.

The SCM Process

Software configuration management is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is also responsible for the identification of individual SCIs and various versions of the software, the auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration. Any discussion of SCM introduces a set of complex questions:

1. How does an organization identify and manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
2. How does an organization control changes before and after software is released to a customer?
3. Who has responsibility for approving and ranking changes?
4. How can we ensure that changes have been made properly?
5. What mechanism is used to appraise others of changes that are made?

These questions lead us to the definition of five SCM tasks:

1. Baseline Identification
2. Version Control
3. Change Control
4. Configuration Auditing
5. Reporting

SOFTWARE ENGINEERING

Baseline Identification

Change is a fact of life in software development.

- Customers want to modify requirements.
- Developers want to modify the technical approach.
- Managers want to modify the project strategy.

Why do we need modification? The answer is really quite simple.

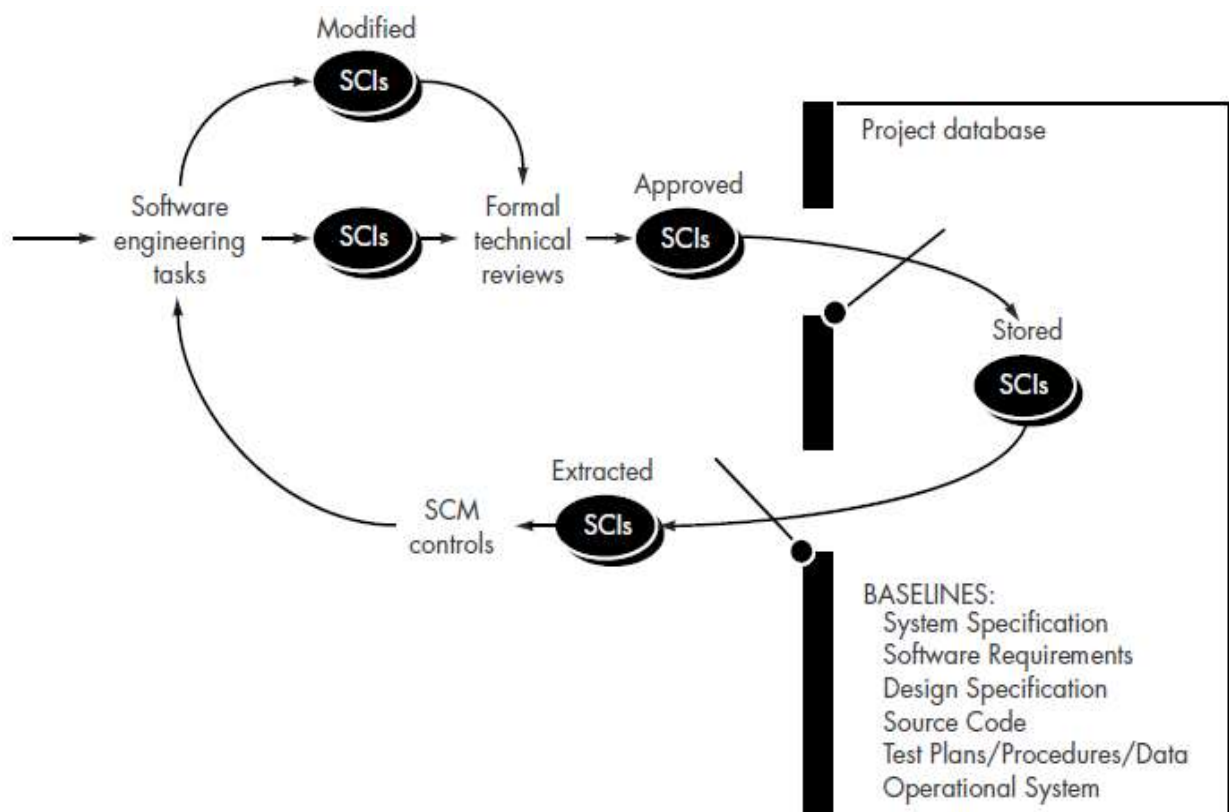
As time passes, all constituencies know more about

- what they need
- which approach would be best
- how to get it done and still make money

This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: Most changes are justified!

IEEE Definition: A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

In the context of software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review.



SOFTWARE ENGINEERING

Version Control

Definition: Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For example if you are a graphic or web designer and want to keep every version of an image or layout for future reference, it is very wise to use a Version Control System (VCS). A VCS allows you to:

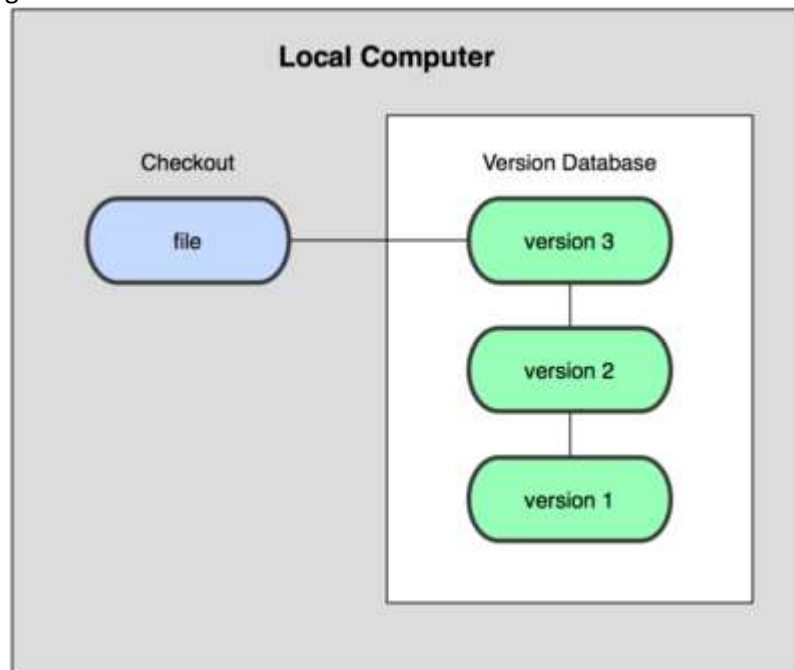
- Revert files back to a previous state
- Revert the entire project back to a previous state
- Review changes made over time
- See who last modified something that might be causing a problem
- Who introduced an issue and when, and more.

Using a VCS also means that if you lose files, you can generally recover easily. In addition, you get all this for very little overhead.

Local Version Control Systems

Local version-control method of choice is to copy files into another directory. This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control



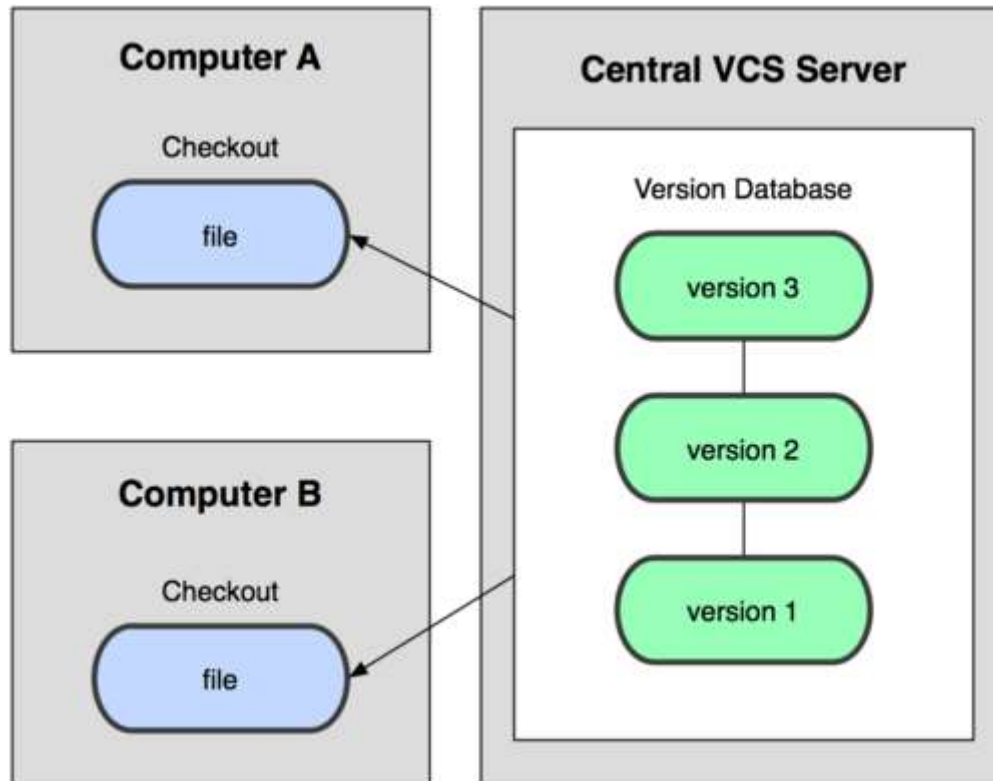
Local version control diagram.

One of the more popular VCS tools was a system called RCS (Revision Control System), which is still distributed with many computers today. This tool basically works by keeping patch sets (that is, the differences between files) from one revision to another in a special format on disk; it can then recreate what any file looked like at any point in time by adding up all the patches.

SOFTWARE ENGINEERING

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCS) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



Centralized version control diagram.

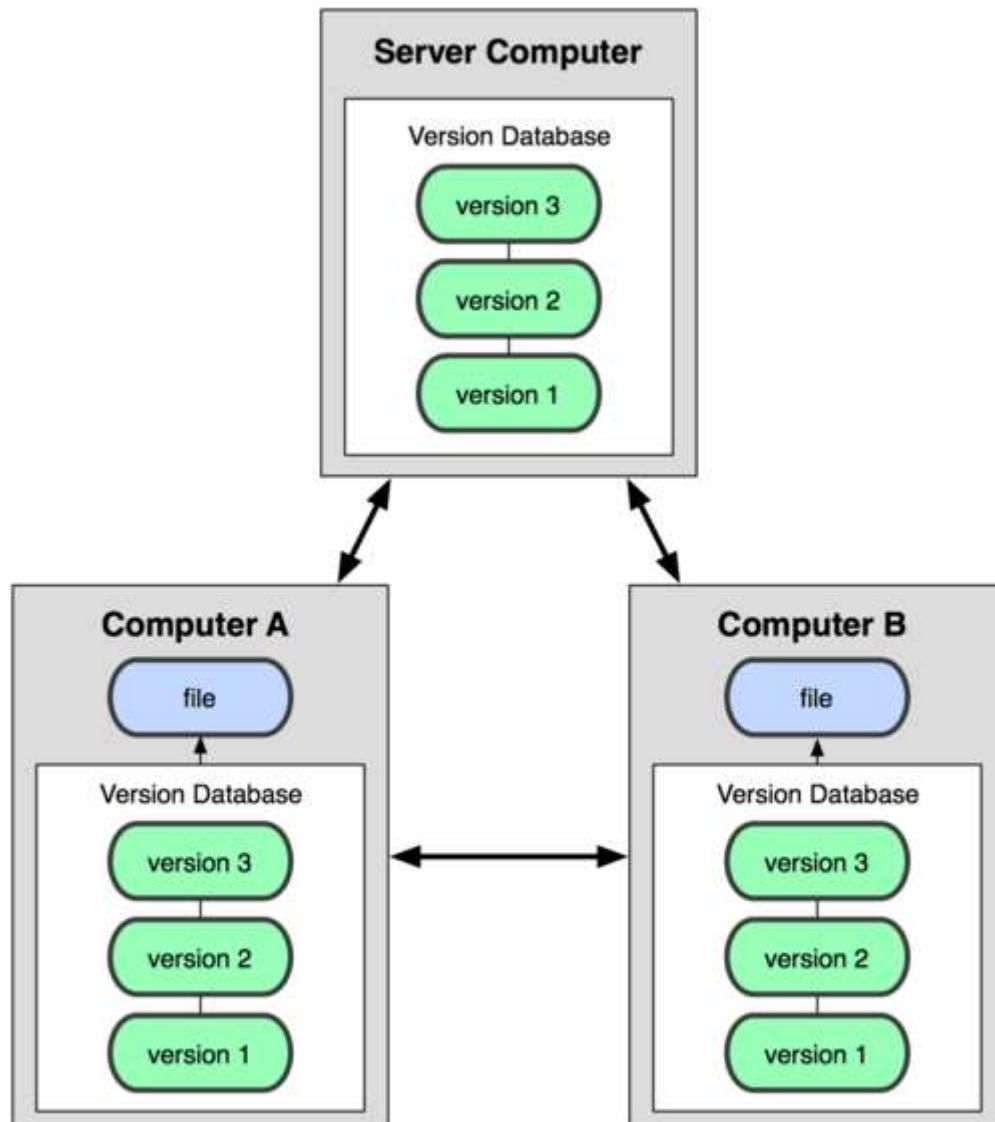
This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything—the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem—whenever you have the entire history of the project in a single place, you risk losing everything.

SOFTWARE ENGINEERING

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every checkout is really a full backup of all the data.



Distributed version control diagram.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.

SOFTWARE ENGINEERING

Change Control

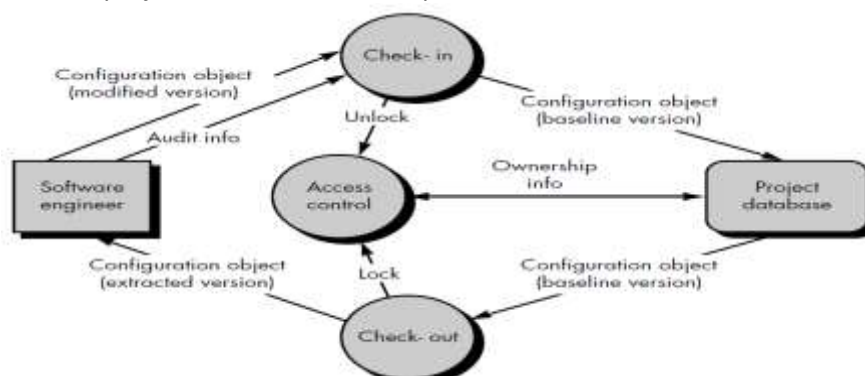
Change control is vital.

- A tiny change in the code can create a big failure in the product.
- It can also fix a big failure or enable wonderful new capabilities.

There should be a balancing act. Too much change control and we create problems. Too little, and we create other problems. For a large software engineering project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change.

The change control process is illustrated as follows –

1. A change request is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change.
2. The results of the evaluation are presented as a change report, which is used by a change control authority (CCA)—a person or group who makes a final decision on the status and priority of the change.
3. An Engineering Change Order (ECO) is generated for each approved change.
4. The ECO describes what change to be made, the constraints that must be respected, and the criteria for review and audit.
5. The object to be changed is "checked out" of the project database, the change is made, and appropriate SQA activities are applied.
6. The object is then "checked in" to the database and appropriate version control mechanisms are used to create the next version of the software.
7. The "check-in" and "check-out" process implements two important elements of change control—access control and synchronization control.
8. Access control governs which software engineers have the authority to access and modify a particular configuration object.
9. Synchronization control helps to ensure that parallel changes, performed by two different people, don't overwrite one another.
10. Based on an approved change request and ECO, a software engineer checks out a configuration object.
11. An access control function ensures that the software engineer has authority to check out the object, and synchronization control locks the object in the project database so that no updates can be made to it until the currently checked out version has been replaced.
12. Note that other copies can be checked-out, but other updates cannot be made. A copy of the baselined object, called the extracted version, is modified by the software engineer.
13. After appropriate SQA and testing, the modified version of the object is checked in and the new baseline object is unlocked.
14. The modified project is then released to production.



SOFTWARE ENGINEERING

Configuration Auditing

How can we ensure that the change has been properly implemented? The answer is twofold:

- Formal technical reviews
- The software configuration audit.

Formal technical reviews

1. The formal technical review focuses on the technical correctness of the configuration object that has been modified.
2. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side
3. effects.
4. A formal technical review should be conducted for all but the most trivial changes.
5. A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review.

The audit asks and answers the following questions:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a formal technical review been conducted to assess technical correctness?
3. Has the software process been followed and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified?
5. Do the attributes of the configuration object reflect the change?
6. Have SCM procedures for noting the change, recording it, and reporting it been followed?
7. Have all related SCIs been properly updated?

Reporting

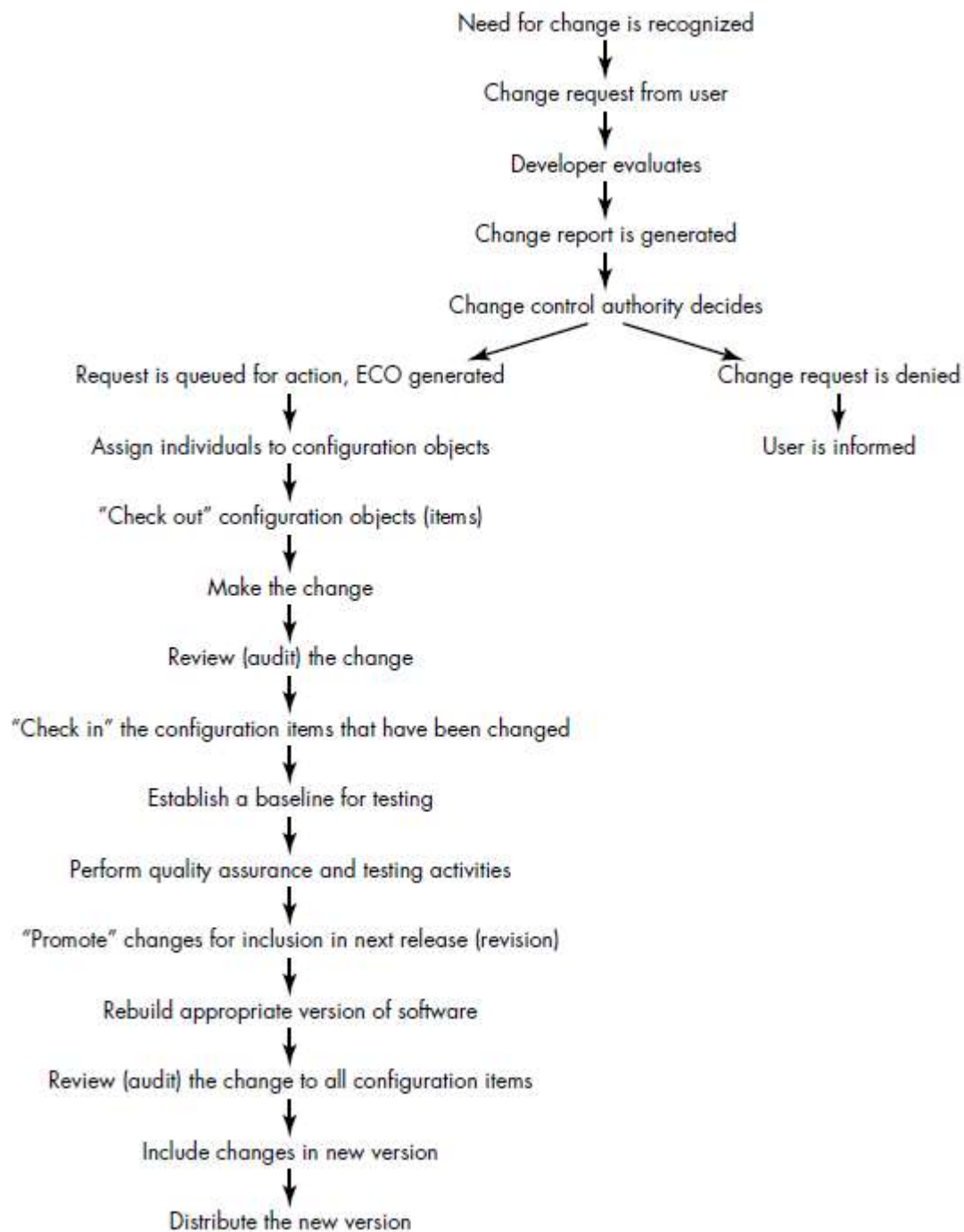
Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions:

1. What happened?
2. Who did it?
3. When did it happen?
4. What else will be affected?

Configuration status reporting plays a vital role in the success of a large software development project:

- Each time an SCI is assigned new or updated identification, a CSR entry is made.
- Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made.
- Each time a configuration audit is conducted, the results are reported as part of the CSR task.
- Output from CSR may be placed in an on-line database, so that software developers or maintainers can access change information by keyword category.
- In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners apprised of important changes.

Configuration Status Reporting (CSR):



SOFTWARE ENGINEERING

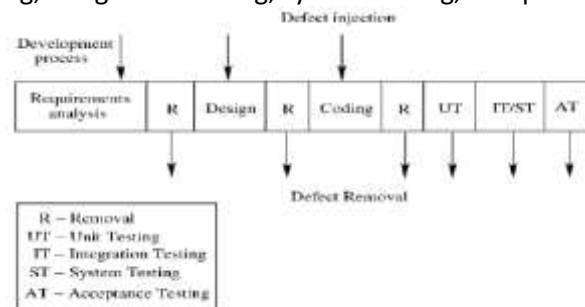
Quality Plan

Having set the goals for effort and schedule, the goal for the third key dimension of a project—quality—needs to be defined. However, unlike schedule and effort, quantified quality goal setting for a project and then planning to meet it is much harder. For effort and schedule goals, we can easily check if a detailed plan meets these goals. For quality, even if we set the goal in terms of expected delivered defect density, it is not easy to plan for achieving this goal or for checking if a plan can meet these goals. Hence, often, quality goals are specified in terms of acceptance criteria—

- The delivered software should finally work for all the situations and test cases in the acceptance criteria.
- There may even be an acceptance criterion on the number of defects that can be found during the acceptance testing.
- The quality plan is the set of quality-related activities that a project plans to do to achieve the quality goal.

Software development is a highly people-oriented activity and hence it is error-prone. To plan for quality, we need to understand the defect injection and removal cycle, as it is defects that determine the quality of the final delivered software.

- Defects are injected into the software being built during the different phases in the project. Through the development phase the software is required to satisfy those needs.
- These injection stages are primarily the requirements specification, the high-level design, the detailed design, and coding.
- To ensure that high-quality software is delivered, these defects are removed through the quality control (QC) activities.
- The QC activities for defect removal include requirements reviews, design reviews, code reviews, unit testing, integration testing, system testing, acceptance testing, etc.



As the final goal is to deliver software with low defect density, ensuring quality revolves around two main themes: reduce the defects being injected, and increase the defects being removed.

- Understand and maintain standards, methodologies, following of good processes, etc., which help reduce the chances of errors by the project personnel.
- Maintain a quality plan focusing on suitable quality control tasks for removing defects.
- Reviews and testing are two most common QC activities utilized in a project.
- Reviews are structured, human-oriented processes to get the current
- Testing is the process of executing software (or parts of it) in an attempt to identify defects.
- Specify the QC activities to be performed in the project, and have suitable guidelines for performing each of the QC tasks.
- During project execution, these activities are carried out in accordance with the defined procedures.
- For quantitative assessment of the quality processes, metrics-based analysis is necessary.
- QC tasks will be schedulable tasks in the detailed schedule of the project. It will specify what documents will be inspected, what parts of the code will be inspected, and what levels of testing will be performed.

SOFTWARE ENGINEERING

Risk Management

Risk management is an attempt to minimize the chances of failure caused by unplanned events. The aim of risk management is not to avoid getting into projects that have risks but to minimize the impact of risks in the projects that are undertaken.

A risk is a probabilistic event—it may or may not occur.

Risk Management is considered first among the best practices for managing large software projects.

Risk Management Concepts

Risk is defined as an exposure to the chance of injury or loss. That is, risk implies that there is a possibility that something negative may happen.

In the context of software projects, negative implies that there is an adverse effect on cost, quality, or schedule. Risk management is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimal.

Risk management can be considered as dealing with the possibility and actual occurrence of those events that are not “regular” or commonly expected, but they are probabilistic. It deals with events that are infrequent, somewhat out of the control of the project management, and which can have a major impact on the project.

The idea of risk management is to minimize the possibility of risks materializing, if possible, or to minimize the effects if risks actually materialize.

Risk management has to deal with identifying the undesirable events that can occur, the probability of their occurring, and the loss if an undesirable event does occur.

Examples:

- In a project there could be an expected risk, such as people going on leave or some requirements change. These are handled by normal project management. So, in a sense, risk management begins where normal project management ends.
- When constructing a building, there is a risk that the building may later collapse due to an earthquake. That is, the possibility of an earthquake is a risk. If the building is a large residential complex, then the potential cost in case the earthquake risk materializes can be enormous. This risk can be reduced by shifting to a zone that is not earthquake-prone.

SOFTWARE ENGINEERING

Risk Assessment

The goal of risk assessment is to prioritize the risks so that attention and resources can be focused on the more risky items. Risk identification is the first step in risk assessment, which identifies all the different risks for a particular project. These risks are project-dependent and identifying them is an exercise in envisioning what can go wrong. Methods that can aid risk identification include checklists of possible risks, surveys, meetings and brainstorming, and reviews of plans, processes, and work products.

List of the top 5 risk items likely to compromise the success of a software project.

	Risk Item	Risk Management Techniques
1	Personnel Shortfalls	Staffing with top talent; Job matching; Team building; Key personnel agreements; Training; Prescheduling key people
2	Unrealistic Schedules and Budgets	Detailed cost and schedule estimation; Design to cost; Incremental development; Software reuse; Requirements scrubbing
3	Developing the Wrong Software Functions	Organization analysis; Machine analysis; User surveys; Prototyping; Early user's manuals
4	Developing the Wrong User Interface	Prototyping; Scenarios; Task analysis; User characterization
5	Gold Plating	Requirements scrubbing; Prototyping; Cost benefit analysis; Design to cost

Risk Control

- The main objective of risk control is to identify the top few risk items and then focus on them.
- Once the risks have been identified and prioritized the risks, the top risks can be easily identified.
- Knowing the risks is of value to prepare a plan to counter their impact.
- One obvious strategy is risk avoidance, which entails taking actions that will avoid the risk altogether.
- For most risks, the strategy is to perform the actions that will either reduce the probability of the risk materializing or reduce the loss due to the risk materializing.
- Risk Control comprises of active measures that have to be performed to minimize the impact of risks.
- Risk prioritization and control are based on the risk perception at the time the risk analysis is performed.
- One simple approach for risk monitoring is to analyze the risks afresh at each major milestone, and change the plans as needed.

Project Monitoring Plan

A project management plan is merely a document that can be used to guide the execution of a project. The main goal of project managers for monitoring a project is to get visibility into the project execution so that they can determine whether any action needs to be taken to ensure that the project goals are met.

As project goals are in terms of effort, schedule, and quality, the focus of monitoring is on these aspects. Different levels of monitoring might be done for a project. The three main levels of monitoring are:

1. Activity level
2. Status reporting
3. Milestone analysis

Activity-level monitoring

- Ensures that each activity in the detailed schedule has been done properly and within time.
- Typically done daily in project team meetings or by the project manager checking the status of all the tasks scheduled to be completed on that day.
- Each completed task is marked as 100%.
- Tools like Microsoft Project is used to track the percentage completion of the overall project or a higher-level task.
- This monitoring is to ensure that the project continues to proceed as per the planned schedule.

Status reports

- Are often prepared weekly to take stock of what has happened and what needs to be done.
- Status reports typically contain
 - a summary of the activities successfully completed since the last status report
 1. any activities that have been delayed
 2. any issues in the project that need attention,
 3. and if everything is in place for the next week.

Milestone Analysis

- Is done at each milestone or every few weeks,
- Analysis of actual versus estimated for effort and schedule is often included in the milestone analysis. If the deviation is significant, it means that the project may run into trouble and might not meet its objectives.
- Project managers try to understand the reasons for the variation and to apply corrective and preventive actions if necessary.
- Defects found by different quality control tasks, and the number of defects fixed may also be reported. This report monitors the progress of the project with respect to all the goals

12. Software Testing

- The development of software systems involves a series of production activities where injection of human error is enormous.
- Errors may be at the very start of the process where the requirements may be imperfectly specified, as well as in later design and development stages.
- Because of human inability to perform and communicate with perfection, software development is accompanied by a quality assurance activity.

Testing Fundamentals

What is Software Testing?

Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. The goal is to design a series of test cases that have a high likelihood of finding errors. This is accomplished using software testing techniques. These techniques provide systematic guidance for designing tests that exercise the internal logic of software components and exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.

Who does Software Testing?

During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved.

Why is Software Testing important?

Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

How to test the Software?

Software is tested from two different perspectives:

1. internal program logic is exercised using “white box” test case design techniques.
2. Software requirements are exercised using “black box” test case design techniques.

In both cases, the intent is to find the maximum number of errors with the minimum amount of effort

and time. When you begin testing, change your point of view. Try hard to “break” the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness.

Black-Box Testing

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

In black-box testing, the tester only knows the inputs that can be given to the system and what output the system should give. In other words, the basis for deciding test cases is the requirements or specifications of the system or module. This form of testing is also called functional or behavioral testing.

SOFTWARE ENGINEERING

The most obvious functional testing procedure is exhaustive testing, which is impractical. The other way is to generate test cases randomly which is less successful. So we need to use techniques or heuristics that can be used to select test cases that have been found to be very successful in detecting errors.

Equivalence Class Partitioning

Since exhaustive testing is not possible it is better to divide the input domain or values into a set of equivalence classes, so that if the program works correctly for a value within the range, then it will work correctly for all the other values in that class. If we can indeed identify such classes, then testing the program with one value from each equivalence class is equivalent to doing an exhaustive test of the program.

An equivalence class is formed of the inputs for which the behavior of the system is specified or expected to be similar. Each group of inputs for which the behavior is expected to be different from others is considered a separate equivalence class

For robust software, we must also consider invalid inputs. That is, we should define equivalence classes for invalid inputs also.

Equivalence classes are usually formed by considering each condition specified on an input as specifying a valid equivalence class and one or more invalid equivalence classes.

Assume that the application accepts an integer in the range 100 to 999 the following partitions are made:

- **Valid Equivalence Class partition:**
 - Numerical value in between 100 and 999
- **Non-valid Equivalence Class partitions:**
 - less than 100
 - more than 999
 - decimal numbers
 - Alphabets/non numeric characters.
 - Special Characters

Boundary Value Analysis

In boundary value analysis, we choose an input for a test case from an equivalence class, such that the input lies at the edge of the equivalence classes. Boundary value test cases are also called “extreme cases.” A boundary value test case is a set of input data that lies on the edge or boundary of a class of input data or that generates output that lies at the boundary of a class of output data. In case of ranges, for boundary value analysis it is useful to select the boundary elements of the range and an invalid value just beyond the two ends (for the two invalid equivalence classes).

Assume that the application accepts an integer in the range 100 to 999 the following partitions are made:

- **Valid Boundary Value Analysis:**
 - 100
 - 999
- **Invalid Boundary Value Analysis:**
 - 99 – Less than the minimum boundary
 - 101 – One more than minimum boundary

SOFTWARE ENGINEERING

- 998 – Less than the maximum boundary
- 1000 – One more than maximum boundary

Pairwise Testing

Most faults tend to be either single-mode or double-mode.

Single-mode faults can be detected by testing for different values of different parameters. So, if there are n parameters for a system, and each one of them can take m different values, then with each test case we can test one different value of each parameter. In other words, we can test for all the different values in m test cases.

For testing for double mode faults, we need not test the system with all the combinations of parameter

values but need to test such that all combinations of values for each pair of parameters are exercised. This is called pairwise testing.

For example, let's say that the system has three parameters with 3 values:

1. A (operating system): Values - a1, a2, a3
2. B (memory size): Values - b1, b2, b3
3. C (browser): Values - c1, c2, c3

The total number of pairwise combinations is $9 * 3 = 27$. The number of test cases, however, to cover all the pairs is much less. A test case consisting of values of the three parameters covers three combinations (of A-B, B-C, and A-C). Hence, in the best case, we can cover all 27 combinations by $27/3=9$ test cases.

These test cases are shown below, along with the pairs they cover.

A	B	C	Pairs
a1	b1	c1	(a1,b1) (a1,c1) (b1,c1)
a1	b2	c2	(a1,b2) (a1,c2) (b2,c2)
a1	b3	c3	(a1,b3) (a1,c3) (b3,c3)
a2	b1	c2	(a2,b1) (a2,c2) (b1,c2)
a2	b2	c3	(a2,b2) (a2,c3) (b2,c3)
a2	b3	c1	(a2,b3) (a2,c1) (b3,c1)
a3	b1	c3	(a3,b1) (a3,c3) (b1,c3)
a3	b2	c1	(a3,b2) (a3,c1) (b2,c1)
a3	b3	c2	(a3,b3) (a3,c2) (b3,c2)

Special Cases

Programs often produce incorrect behavior when inputs form some special cases. The reason is that in programs, some combinations of inputs need special treatment, and providing proper handling for these special cases is easily overlooked.

For example, in an arithmetic routine, if there is a division and the divisor is zero, some special action has to be taken, which could easily be forgotten by the programmer. These special cases form particularly good test cases, which can reveal errors that will usually not be detected by other test cases.

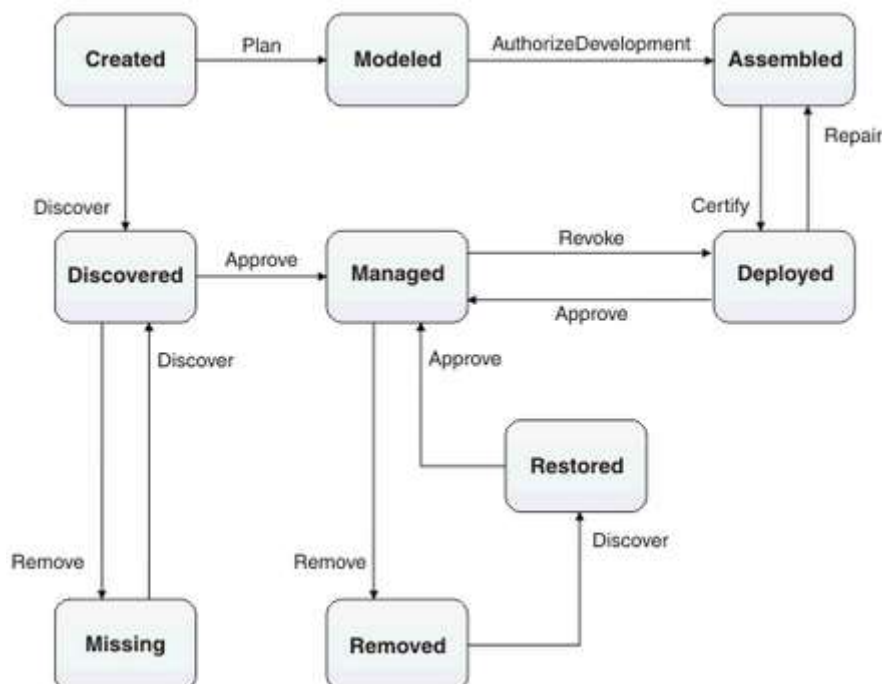
Special cases will often depend on the data structures and the function of the module. There are no rules to determine special cases, and the tester has to use his intuition and experience to identify such test cases. Consequently, determining special cases is also called error guessing.

State-Based Testing

- state: abstract situation in the life cycle of a system entity (for instance, the contents of an object)
- event: a particular input (for instance, a message or method call)
- action: the result, output or operation that follows an event
- transition: an allowable two-state sequence, that is, a change of state ("firing") caused by an event
- guard: predicate expression associated with an event, stating a Boolean restriction for a transition to fire

There are several types of state machines:

- infinite automaton (no guards or actions)
- Mealy machine (no actions associated with states)
- Moore machine (no actions associated with transitions)
- state chart (hierarchical states: common super states)
- state transition diagram: graphic representation of a state machine
- state transition table: tabular representation of a state machine



SOFTWARE ENGINEERING

White-Box Testing

White-box testing is concerned with testing the implementation of the program. The purpose of this testing is not to exercise all the different input or output conditions but to exercise the different programming structures and data structures used in the program. It is also called structural testing.

In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs.

White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently.

White-box test design techniques include the following code coverage criteria:

- Control flow testing
- Data flow testing
- Statement coverage
- Decision coverage
- Path testing

Control flow testing

Most traditional form of white-box testing

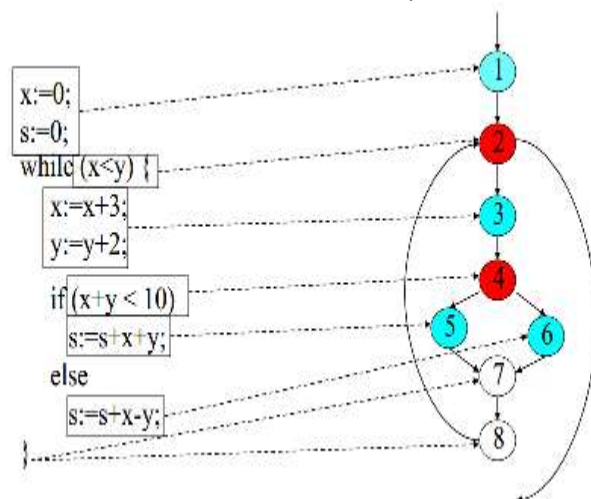
- 1) From the source, create the control flow graph (CFG): it describes the flow of control extracted automatically from source
- 2) Define a "coverage target" over the CFG -- Nodes, edges, paths
- 3) Design test cases to cover "coverage target"

Example:

a. Define Blocks in the code:

```
x:=0;  
s:=0;  
while (x<y) {  
  x:=x+3;  
  y:=y+2;  
  if (x+y < 10)  
    s:=s+x+y;  
  else  
    s:=s+x-y;  
}
```

b. Create Control Flow Graph:



SOFTWARE ENGINEERING

Data flow testing

Data flow is an abstract representation of the sequence and possible changes of state of data objects, where the state of an object is any of creation (created); used: used or modified; destruction (killed). Using Data flow, one can understand how the data acts as they are transformed by the program and also, defects like referencing a variable with an undefined value and variables that are never used can be identified.

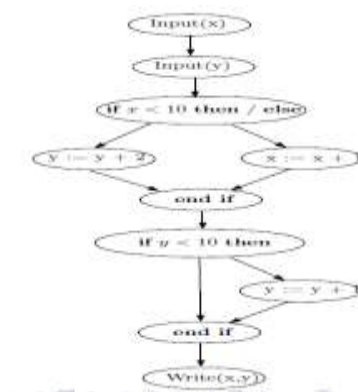
Data flow testing strategies are family of test strategies to track program's control flow in order to explore sequences of events related to status of data objects caused by creation, usage, modification or destruction with the intention of identifying any data anomalies.

Can reveal interesting bugs

- A variable that is defined but never used
- A variable that is used but never defined
- A variable that is defined twice before it is used
- Sending a modifier message to an object more than once between accesses
- De-allocating a variable before it used

Example:

```
1: Input(x)
2: Input(y)
3: if x < 10 then
4:   y := y + 2
5: else
6:   x := x + 1
7: end if
8: if y < 10 then
9:   y := y + 1
10: end if
11: Write(x,y)
12: end
```



Statement coverage

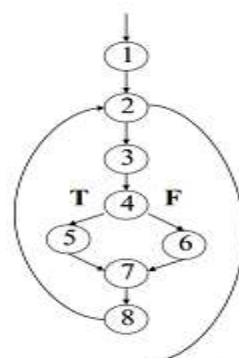
Basic idea: given the control flow graph define a “coverage target” and write test cases to achieve it

- Traditional target: statement coverage
- Need to write test cases that cover all nodes in the control flow graph
- Intuition: code that has never been executed during testing may contain errors

Example:

Suppose that we write and execute two test cases

- Test case #1: follows path 1-2-exit (NOTE: never take the loop)
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
- Test case #3: 1-2-3-4-6-7-8-2-3-4-6-7-8-2-exit



SOFTWARE ENGINEERING

Decision coverage

Decision Coverage is used whenever there are two or more possible exits from the statement like an IF statement, a DO-WHILE or a CASE statement which could be either TRUE or FALSE.

Decision coverage ensures that each outcome (i.e. TRUE and FALSE) of control statement has been executed at least once. Alternatively you can say that control statement IF has been evaluated both to TRUE and FALSE.

The formula to calculate decision coverage is:

Decision Coverage= (Number of decision outcomes executed/Total number of decision outcomes)*100%

Decision coverage is stronger than statement coverage and it requires more test cases to achieve 100% decision coverage.

Let us take one example to explain decision coverage:

```
READ X
READ Y
IF "X > Y"
PRINT X is greater than Y
ENDIF
```

To get 100% statement coverage only one test case is sufficient for this pseudo-code.

TEST CASE 1: X=10 Y=5

However this test case won't give you 100% decision coverage as the FALSE condition of the IF statement is not exercised.

In order to achieve 100% decision coverage we need to exercise the FALSE condition of the IF statement which will be covered when X is less than Y.

So the final TEST SET for 100% decision coverage will be:

TEST CASE 1: X=10, Y=5
TEST CASE 2: X=2, Y=10

Note: 100% decision coverage guarantees 100% statement coverage but 100% statement coverage does not guarantee 100% decision coverage.

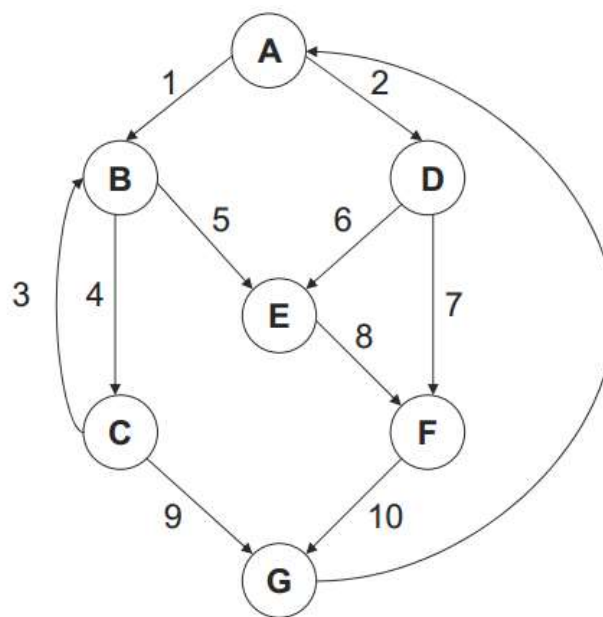
Path testing

Path testing is a testing strategy that aims to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once. All conditional statements are tested for both true and false cases. In an object-oriented development process, path testing may be used when testing the methods associated with objects.

Some of the most general guidelines that are:

- Choose inputs that force the system to generate all error messages;
- Design inputs that cause input buffers to overflow;
- Repeat the same input or series of inputs numerous times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small.

The starting point for path testing is a program flow graph. This is a skeletal model of all paths through the program. A flow graph consists of nodes representing decisions and edges showing flow of control. The flow graph is constructed by replacing program control statements by equivalent diagrams.



Testing process

The basic goal of the software development process is to produce software that has no errors or very few errors. Testing is a quality control activity which focuses on identifying defects (which are then removed). Different levels of testing are needed to detect the defects injected during the various tasks in the project.

And at a level multiple SUT's (Software under Test) may be tested. And for testing each SUT, test cases will have to be designed and then executed.

Test Plan

In general, in a project, testing commences with a test plan and terminates with successful execution of user acceptance testing. A test plan is a general document for the entire project that defines

- The scope
- Approach to be taken
- The schedule of testing
- Identifies the test items for testing
- Identifies the people responsible for the different activities of testing.

The test planning can be done well before the actual testing commences and can be done in parallel with the coding and design activities. The inputs for forming the test plan are:

- Project plan
- Requirements document
- Architecture or design document.

The project plan is needed to make sure that the test plan is consistent with the overall quality plan for the project and the testing schedule matches that of the project plan. The requirements document and the design document are the basic documents used for selecting the test units and deciding the approaches to be used during testing. A test plan should contain the following:

- Test unit specification
- Features to be tested
- Approach for testing
- Test deliverables
- Schedule and task allocation

As mentioned earlier, different levels of testing have to be performed in a project. The levels are specified in the test plan by identifying the test units for the project. A test unit is a set of one or more modules that form a software under test (SUT). The identification of test units establishes the different levels of testing that will be performed in the project.

Features to be tested include all software features and combinations of features that should be tested. A software feature is a software characteristic specified or implied by the requirements or design documents. These may include -

- Functionality
- Performance
- Design constraints and attributes.

SOFTWARE ENGINEERING

The approach for testing specifies the overall approach to be followed in the current project. The techniques that will be used to judge the testing effort should also be specified. This is sometimes called the testing criterion or the criterion for evaluating the set of test cases used in testing.

Testing deliverables should be specified in the test plan before the actual testing begins. Deliverables could be a list of -

- Test cases that were used
- Detailed results of testing
- List of defects found
- Test summary report
- Data about the code coverage

The test plan typically also specifies the schedule and effort to be spent on different activities of testing, and the tools to be used. This schedule should be consistent with the overall project schedule. The detailed plan may list all the testing tasks and allocate them to test resources that are responsible for performing them.

Test Case Design

The test plan focuses on how the testing for the project will proceed, which units will be tested, and what approaches (and tools) are to be used during the various stages of testing. However, it does not deal with the details of testing a unit, nor does it specify which test cases are to be used.

Test case design has to be done separately for each unit. Based on the approach specified in the test plan, and the features to be tested, the test cases are designed and specified for testing the unit. Test case specification gives, for each unit to be tested, all test cases, inputs to be used in the test cases, conditions being tested by the test case, and outputs expected for those test cases. If test cases are specified in a document, the specifications look like a table of the form shown below-

Test Case Example1 (simple test)

Test Case #: 2.2	Test Case Name: Change PIN	Page: 1 of 1
System: ATM	Subsystem: PIN	
Designed by: ABC	Design Date: 28/11/2004	
Executed by:	Execution Date:	
Short Description: Test the ATM Change PIN service		

Pre-conditions

The user has a valid ATM card - The user has accessed the ATM by placing his ATM card in the machine
The current PIN is 1234
The system displays the main menu

Step	Action	Expected System Response	Pass/Fail	Comment
1	Click the 'Change PIN' button	The system displays a message asking the user to enter the new PIN		
2	Enter '5555'	The system displays a message asking the user to confirm (re-enter) the new PIN		
3	Re-enter '5555'	The system displays a message of successful operation The system asks the user if he wants to perform other operations		
4	Click 'YES' button	The system displays the main menu		
5	Check post-condition 1			

Post-conditions

1. The new PIN '5555' is saved in the database

Test Case Execution

With the specification of test cases, the next step in the testing process is to execute them. This step is also not straightforward. The test case specifications only specify the set of test cases for the unit to be tested. However, executing the test cases may require construction of driver modules or stubs. It may also require modules to set up the environment as stated in the test plan and test case specifications. Only after all these are ready can the test cases be executed.

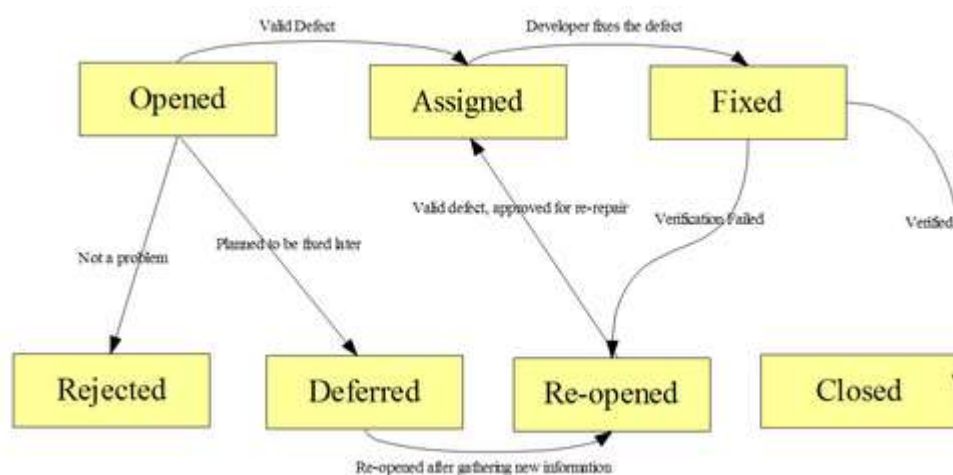
During test case execution, defects are found. These defects are then fixed and testing is done again to verify the fix. To facilitate reporting and tracking of defects found during testing (and other quality control activities), defects found are often logged in a Defect Tracking Software. Defect logging is particularly important in a large software project which may have hundreds or thousands of defects that are found by different people at different stages of the project. Often the person who fixes a defect is not the person who finds or reports the defect.

For example, a tester may find the defect while the developer of the code may actually fix it. In such a scenario, defect reporting and closing cannot be done informally. The use of informal mechanisms may easily lead to defects being found but later forgotten, resulting in defects not getting removed or in extra effort in finding the defect again. Hence, defects found must be properly logged in a system and their closure tracked. Defect logging and tracking is considered one of the best practices for managing a project, and is followed by most software organizations.

Defect Life Cycle

Defect Life Cycle (Bug Life cycle) is the journey of a defect from its identification to its closure. The Life Cycle varies from organization to organization and is governed by the software testing process the organization or project follows and/or the Defect tracking tool being used.

In the figure shown below all the defect reports move through a series of clearly identified states.



States in a Defect Life Cycle:

1. A defect is in open state when the tester finds any variation in the test results during testing, peer tester reviews the defect report and a defect is opened.
2. Now the project team decides whether to fix the defect in that release or to postpone it for future release. If the defect is to be fixed, a developer is assigned the defect and defect moves to assigned state.
3. If the defect is to be fixed in later releases it is moved to deferred state.

SOFTWARE ENGINEERING

4. Once the defect is assigned to the developer it is fixed by developer and moved to fixed state, after this an e-mail is generated by the defect tracking tool to the tester who reported the defect to verify the fix.
5. The tester verifies the fix and closes the defect, after this defect moves to closed state.
6. If the defect fix does not solve the issue reported by tester, tester re-opens the defect and defect moves to re-opened state. It is then approved for re-repair and again assigned to developer.
7. If the project team defers the defect it is moved to deferred state, after this project team decides when to fix the defect. It is re-opened in other development cycles and moved to re-opened state.
8. It is then assigned to developer to fix it.