A TEXT BOOK OF

SOFTWARE ENGINEERING

FOR
SEMESTER – VII

FINAL YEAR B. TECH COURSE IN COMPUTER ENGINEERING

Strictly According to New Revised Credit System Syllabus of Dr. Babasaheb Ambedkar Technological University (DBATU), Lonere, (Dist. Raigad) Maharashtra,
(w.e.f. June 2020-21)

Mrs. AMITA A JAJOO
M.E. (Computer),
Assistant Professor,
Information Technology Dept.,
D.Y. Patil College of Engineering,
Akurdi, PUNE.

KALYAN D. BAMANE
M.E. (Computer),
Assistant Professor,
Information Technology Dept.,
D.Y. Patil College of Engineering,
Akurdi, PUNE.

Price ₹ 150.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N1158

SYLLABUS

Unit 1 : Introduction **[6 Hrs]**

Professional software development, Software engineering ethics, Case studies.

Software processes: Software process models, Process activities, Coping with change, The rational unified process.

Unit 2 : Agile Software Development **[6 Hrs]**

Agile methods, Plan-driven and agile development, Extreme programming, Agile project management, Scaling agile methods.

Requirements Engineering : Functional and non-functional requirements, The software requirements document, Requirements specification, Requirements engineering processes, Requirements elicitation and analysis, Requirements validation, Requirements management.

Unit 3 : System Modeling **[6 Hrs]**

Context models, Interaction models, Structural models, Behavioral models, Model-driven engineering. Architectural design: Architectural design decisions, Architectural views, Architectural patterns, Application architectures.

Unit 4 : Design and Implementation **[6 Hrs]**

Object-oriented design using UML, Design patterns Implementation issues, Open source development.

Unit 5 : Testing **[6 Hrs]**

Software testing, Development testing, Test-driven development, Release testing, User testing.

Unit 6 : **[6 Hrs]**

Dependability properties, Availability and reliability, Safety Security.

CONTENTS

UNIT I

INTRODUCTION

1.1 NATURE OF SOFTWARE

Traditional definition of software is the set of instructions or more generally a collection of computer programs which tell the hardware how to work and perform different tasks on a computer system.

- Software is both a product and a vehicle for delivering a product (information).
- Software is engineered not manufactured.
- Software does not wear out, but it does deteriorate.
- Industry is moving toward component-based software construction, but most software is still custom-built.
- Software is designed and built by software engineers.
- Software is used by virtually everyone in society.
- Software is pervasive in our commerce, our culture, and our everyday lives.
- Software engineers have a moral obligation to build reliable software that does no harm to other people.
- Software engineers view computer software, as being made up of the programs, documents, and data required to design and build the system.
- Software users are only concerned with whether or not software products meet their expectations and make their tasks easier to complete.

Software Characteristics

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware :

- Software costs are concentrated in engineering (analysis and design) and not in production.
- Cost of software is not dependent on volume of production.
- Software does not wear out (in the physical sense).
- Software has no replacement (spare) parts.
- Software maintenance is a difficult problem and is very different from hardware (physical product) maintenance.
- Most software are custom-built, rather than being assembled from existing components..
- Many legal issues are involved (e.g. intellectual property rights, liability).

Characteristics of Real-World Software

- It is generally developed by software firms for their clients under formal business contracts.
- Like any product, software is designed based on some software specification.
- It is usually developed in teams and not by individuals.
- It generally includes clear and detailed documentation (i.e. design manual and users' manual).
- It is meant for users who need not have good knowledge of computers.
- It generally has user-friendly interfaces so that users having limited expertise in computers can operate the system.
- Generally, software is designed to be run on different platforms.
- It has a lifetime in years, after which it becomes obsolete. Hence, software is designed keeping its intended life and cost in view.
- It generally requires some modification from time to time to accommodate changes taking place in the organization and the environment.
- It is developed under formalized product reviews (quality assurance) and formalized testing procedures.
- The cost of software failure may amount to an economic catastrophe. Hence, software is designed for utmost reliability.
- A computer system is prone to misuse or sabotage by persons having ulterior motives from within or from outside the organization. Hence, software is designed to be tamper-proof and protected from misuse or damage.
- Ethical issues (like protecting privacy) are also taken into consideration in designing software.

1.1.1 Software Application Domains

1. System Software

System software is designed to operate the computer hardware, to provide basic functionality, and to provide a platform for running application software. System software includes:

- Operating system, an essential collection of computer programs that manages resources and provides common services for other software. Supervisory programs, boot loaders, shells and window systems are core parts of operating systems. In practice, an operating system comes bundled with additional software (including application software) so that a user can potentially do some work with a computer that only has an operating system.
- Device driver, a computer program that operates or controls a particular type of device that is attached to a computer. Each device needs at least one corresponding device driver; thus a computer needs more than one device driver.
- Utilities, software designed to assist users in maintenance and care of their computers.

2. Application Software

- It performs productive tasks for user such as word processing and database management etc. An application is a program, or group of programs, that is designed for the end user, for example database programs, word processors, Web browsers and spreadsheets.

3. Real-Time Software

- Software that monitors/analyzes/controls real-world events as they occur is called real time. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

4. Business Software

- Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into Management Information System (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making.

5. Engineering and Scientific Software

- Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

6. Embedded Software

- Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

7. Artificial Intelligence Software

- Artificial Intelligence (AI) software makes use of non numerical algorithms to solve complex problems that are not acquiescent to computation or straight forward analysis. Expert systems, also called knowledge based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of application.

1.1.2 Web-Apps

It has following special characteristics :

- **Network Intensive :** Web App resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., Internet) or more limited access and communication (i.e., corporate intranet).
- **Concurrency :** A large number of users may access the Web-App at any time. In many cases the patterns of usage among end users will vary greatly.
- **Unpredictable Load :** The number of users of the Web-App may vary by orders of magnitude from day to day. For example: 100 users may show up on Monday; 10,000 users use the system on Thursday.

- **Performance :** If a Web-App user must wait too long, he/she may decide to go elsewhere.
- **Availability :** Users of popular Web-Apps often demand access on a 24/7/365 basis.
- **Data Driven :** The primary function of many Web-Apps is to use hypermedia to present text, graphics, audio and video contents to the end user. In addition, Web-Apps are commonly used to access information that exists on data bases that are not an integral part of the web based environment. For example: e-commerce or financial applications.
- **Content Sensitive :** The quality and aesthetic nature of content remains an important determinant of the quality of a Web-App.
- **Continuous Evolution :** Web applications evolve continuously. It is not unusual for some Web-Apps to be updated on a minute-by-minute schedule.
- **Immediacy (Short Time to Market) :** Although immediacy the compelling needs to get software to market quickly is a characteristic of many application domains, Web-Apps often exhibit a time-to-market that can be a matter of a few days or weeks.
- **Security :** In order to protect sensitive content and provide secure modes of f data transmission, strong security measures must be implemented throughout the infrastructure that supports a Web-App and within the application itself.
- **Aesthetics :** Look and feel of Web-App must be attractive.

1.1.3 Mobile Apps

- It is a term used to describe Internet applications that run on smart phones and other mobile devices. Mobile applications usually help users by connecting them to Internet services more commonly accessed on desktop or notebook computers, or help them by making it easier to use the Internet on their portable devices. A mobile app may be a mobile Web site book marking utility, a mobile-based instant messaging client, Gmail for mobile, WhatsApp and many other applications.
- It has following characteristics :

1. **Connectivity**

- The device is continuously logged into mobile network as the apps are constantly online. In this way users particular information and notifications being pushed to app as they are accessible. This is an important characteristic of mobile application to be available anywhere. The ability of push becomes essential in growing apps on every Smartphone as this characteristic keep the app in user minds.

2. **Localization**

- Localize information and the opportunity to provide position based information is critical feature that craft mobility stunning and practical.

3. **Convenience**

- A simple and emotional design guarantees high value and acceptance among users. The overall usability and information structure must be planned carefully to build a joyful and fit interaction flow.

4. **Approachable**

- Approachability covers additional attribute provided by the mobile apps. An excellent application can really be used any time at any place as reach ability becomes full time availability.

5. **Individualization or Personalization**

- Creating individualized content based on personalized context or usage is another feature. Everyone wish that my application should fit my needs and behave like what I want to do. This specific feature not only covers personalized content but wants to control over shared and store data for more actions.

6. **Security**

- There are several aspects of security like transferred data over network via carrier network. Some applications connect data with online, web apps, so the storage of the information on server must be secure. Another critical security break can be the mobile device itself as no one wants anybody to play with his/her mobile phone.

1.1.4 Cloud Computing

- Cloud Computing is a general term used to describe a new class of network based computing that takes place over the Internet :
  - ➢ A collection/group of integrated and networked hardware, software and Internet infrastructure (called a platform).
  - ➢ Using the Internet for communication and transport provides hardware, software and networking services to clients.

- These platforms hide the complexity and details of the underlying infrastructure from users and applications by providing very simple graphical interface or API (Applications Programming Interface).
- In addition, the platform provides on demand services that are always on anywhere, anytime and anyplace.
- Pay for use and as needed, elastic scale up and down in capacity and functionalities.
- The hardware and software services are available to general public, enterprises, corporations and businesses markets.

Basic Cloud Characteristics

- The "no-need-to-know" in terms of the underlying details of infrastructure, applications interface with the infrastructure via the APIs.
- The "flexibility and elasticity" allows these systems to scale up and down at :
  - ➢ Utilizing the resources of all kinds
  - ➢ CPU, storage, server capacity, load balancing, and databases
- The "pay as much as used and needed" type of utility computing and the "always on!, anywhere and any place" type of network-based computing.
- Cloud are transparent to users and applications, they can be built in multiple ways :
  - ➢ Branded products, proprietary open source, hardware or software, or just off-the-shelf PCs.
- In general, they are built on clusters of PC servers and off-the-shelf components plus open source software combined with in-house applications and/or system software.

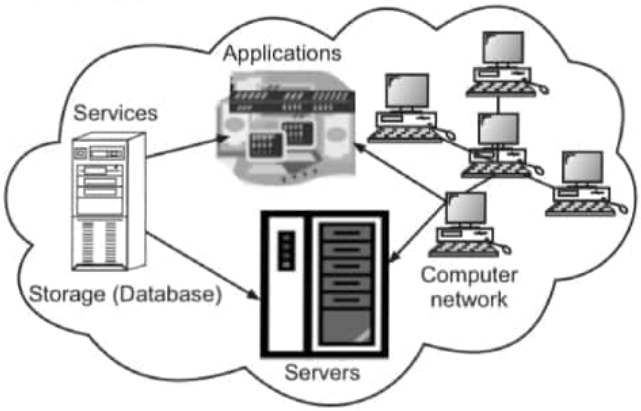


Fig. 1.1 : Cloud Computing

Advantages of Cloud Computing

- **Lower Computer Costs :** No need of a high-powered and high-priced computer to run cloud computing's web-based applications.
- **Improved Performance :** Computers in a cloud computing system boot and run faster because they have fewer programs and processes loaded into memory.
- **Reduced Software Costs :** Instead of purchasing expensive software applications, we can use it from cloud.
- **Instant Software Updates :**When we access a web-based application, we get the latest version and without needing to pay for or download an upgrade.
- **Unlimited Storage Capacity :** Cloud computing offers virtually limitless storage.
- **Increased Data Reliability :** Unlike desktop computing, in which if a hard disk crashes and destroy all valuable data, a computer crashing in the cloud should not affect the storage of data.

1.1.5 Product Line Software

- A Software Product Line (SPL) is a set of software intensive systems that share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.
- Software Product Line (SPL) engineering refers to the engineering and management techniques to create, evolve, and sustain a software product line.
- **Product Line Approach :** A system of software production that uses reusable software related assets to modify, assemble, instantiate, or generate a line of software or software intensive products.
- **Product Line Architecture :** Description of the structural properties for building a group of related systems (i.e., product line), typically the components and their interrelationships. The inherent guidelines about the use of components must capture the means for handling required variability among the systems (sometimes called reference architecture).

  For Example, Microsoft Office supports Macintosh.
- The Product Line's commonalities and variabilities are described in the Problem Space. This reflects the desired range of applications ("product variants") in

the Product Line (the "domain") and their inter-dependencies. So, when producing a product variant, the application developer uses the problem space definition to describe the desired combination of problem variabilities to implement the product variant.

- An associated Solution Space describes the constituent assets of the Product Line (the "platform") and its relation to the problem space, i.e. rules for how elements of the platform are selected when certain values in the problem space are selected as part of a product variant. The four-part division resulting from the combination of the problem space and solution space with domain and application engineering is shown below.

	Problem Space	Solution Space
Domain Engineering	Variability within problem area	Structure and selection rules for solution elements of the Product Line Platform
Application engineering	Specification of product variant	Needed platform and application software

1.2 SOFTWARE ENGINEERING

- Software engineering is the establishment of sound engineering principles in order to obtain reliable and efficient software in an economical manner.
- Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
- Software engineering encompasses a process, management techniques, technical methods, and the use of tools.

Software Engineering Realities

- Problem should be understood before software solution is developed.
- Design is a key activity.
- Software should exhibit high quality.
- Software should be maintainable.

1.2.1 The Discipline

Software engineering can be divided into ten sub disciplines.

1. **Software Requirements :** The elicitation, analysis, specification, and validation of requirements for software.
2. **Software Design :** The process of defining the architecture, components, interfaces, and other characteristics of a system or component. It is also defined as the result of that process.
3. **Software Construction :** The detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.
4. **Software Testing :** The dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.
5. **Software Maintenance :** The totality of activities required to provide cost-effective support to software.
6. **Software Configuration Management :** The identification of the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the system life cycle.
7. **Software Engineering Management :** The application of management activities planning, coordinating, measuring, monitoring, controlling, and reporting to ensure that the development and maintenance of software is systematic, disciplined, and quantified.
8. **Software Engineering Process :** The definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle process itself.
9. **Software Engineering Tools and Methods :** The computer-based tools that are intended to assist the software life cycle processes see Computer Aided Software Engineering, and the methods which impose structure on the software engineering activity with the goal of making the activity systematic and ultimately more likely to be successful.
10. **Software Quality :** The degree to which a set of inherent characteristics fulfills requirements.

The Software Crisis

- As software's importance grew, the software community was always attempting to develop technologies that would make it easier, faster and less expensive to do thing. As software houses emerged, with solutions to everyday problems, software was used in diverse areas. It was used to ease or replace

many unexciting activities like data entry and pay slip generation. It began to be developed for widespread distribution. The software development projects produced ten thousands of source programs. With no tools or methods for managing this increase in complexity, the estimation also went haywire. The time estimated for project completion and the cost estimation made were completely off the mark. This naturally increases the cost, as time meant money.

- With software being so widely used, customer perspective also changed, people got used to expecting more from software. Since the customers had a lot of expectations from the software that they were paying for, the customer could not understand why the software was failing!
- The demand for software was increase and so were the problems. Problems with the software were perceived to be mainly four :

1. **It's Always Late!**

   The development of software always overshot the initial estimates.

2. **It's Over Budget!**

   The cost to create the software was higher than the initial estimates. Most of the time there were huge variation.

3. **It's Buggy!**

   The errors and defects led to definite failure of the software. The software which was being delivered to the customers was bound to fail at some point of time.

4. **Customers were Not Satisfied!**

   The defects were usually found by the customers and these were difficult to track and fix.

- The software companies were losing face their customers. Even those projects that were completed successfully were either late or way out of the estimated budget. A NATO conference was organized and there these problems were collectively referred to as the "Software Crisis". Some characteristics of the software scenario were described as :
  - ➢ Hardware sophistication outpaced the ability to build software that completely taps into its potential.
  - ➢ The ability to build software could not keep pace with the demand for new software.
  - ➢ The ability to maintain existing programs was undermined by poor design and haphazard development practices.

1.2.2 Software Engineering Layers

- Software engineering is a layered technology. Software engineering is divided into four layers
  1. A quality focus
  2. Process
  3. Methods
  4. Tools

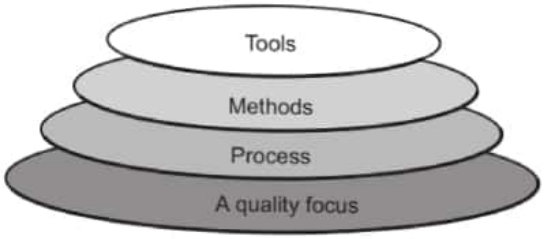


Fig. 1.2 : Software Engineering Layers

1. A Quality Focus

- Total quality management promotes continuous process improvement. Quality of software can be measure by:
  - ➢ **Correctness :** Correctness is the degree to which the software performs its required function.
  - ➢ **Maintainability :** Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change.
  - ➢ **Integrity :** Software integrity has become increasingly important in the age of hackers and firewalls. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security.
  - ➢ **Usability :** Usability is an attempt to quantify user-friendliness; user must be able to operate software easily.

2. Process

- Process defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology.
- The key process areas form the basis for management control of software projects and establish the context

in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

3. Methods

- Software engineering methods provide the technical how-to's for building software.
- Methods encompass a broad array of tasks that include :
  - ➢ Requirements analysis.
  - ➢ Analysis and design modeling.
  - ➢ Program construction.
  - ➢ Testing, and support.

4. Tools

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

  For Example : Star UML, Rational Rose can be used as Object Oriented Analysis and Design tool.

1.2.3 Software Myths

- Software myths propagate misinformation, mis-understanding or confusion in software development process which causes serious problems.
- There are myths at following three levels :
  1. Management level myths.
  2. Customer level myths.
  3. Practitioner level myths.

1. Management Level Myths

Managers with software responsibility are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

- **Myth :** We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?
- **Reality :** The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."
- **Myth :** My people have state-of-the-art software development tools, after all, we buy them the newest computers.
- **Reality :** It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-Aided Software Engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.
- **Myth :** If we get behind schedule, we can add more programmers and catch up.
- **Reality :** Software development is not a mechanistic process like manufacturing.

  As new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.
- **Myth :** If I decide to outsource the software project to a third party, I can just relax and let that firm build it.
- **Reality :** If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

2. Customer Level Myths

- **Myth :** A general statement of objectives is sufficient to begin writing programs the details can be given later on.
- **Reality :** A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.
- **Myth :** Project requirements continually change, but change can be easily accommodated in software.
- **Reality :** It is true that software requirements change, but the impact of change varies with the But if change is requested in later stages, cost of change rapidly increases. If change id requested at maintenance stage all design, coding, testing has to be modified.

3. Practitioner Level Myths

- **Myth :** Once we write the program and get it to work, our job is done.
- **Reality :** Almost 60 to 80 percent work is requires after delivering the project to the customer for the first time.
- **Myth :** Until I get the program "running" I have no way of assessing its quality.
- **Reality :** The most effective software quality assurance mechanisms can be applied from the inception of a project the formal technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.
- **Myth :** The only deliverable work product for a successful project is the working program.
- **Reality :** A working program is only one part of a software product. Actual project delivery includes documentation and guidance for software support.
- **Myth :** Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.
- **Reality :** Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

1.3 PROFESSIONAL SOFTWARE DEVELOPMENT

1.3.1 Essential Attributes of Good Software

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

In Order to do Software Development Well we Need :

- **A Small, Well Integrated Team:** Small teams have fewer lines of communication than larger ones. It's easier to get to know your teammates on a small team. You can get to know your teammates' strengths and weaknesses, who knows what, and who is the "go to guy" for particular problems or particular tools. Well-integrated teams have usually worked on several projects together. Keeping a team together across several projects is a major job of the team's manager. Well-integrated teams are more productive, they are better at holding to a schedule, and they produce code with fewer defects at release. The key to keeping a team together is to give them interesting work to do and then leave them alone.
- **Good Communication Among Team Members:** Constant communication among team members is critical to day-to-day progress and successful project completion. Teams that are co-located are better at communicating and communicate more than teams that are distributed geographically (even if they're just on different floors or wings of a building). This is a major issue with larger companies that have software development sites scattered across the globe.
- **Good Communication Between the Team and the Customer:** Communication with the customer is essential to controlling requirements and requirements churn during a project. On-site or close-by customers allow for constant interaction with the development team. Customers can give immediate feedback on new releases and be involved in creating system and acceptance tests for the product.
- **A Process that Everyone Buys Into:** Every project, no matter how big or small, follows a process. Larger projects and larger teams tend to be more plan-driven and follow processes with more rules and documentation required. Larger projects do require more coordination and tighter controls on communication and configuration management. Smaller projects and smaller teams will, these days, tend to follow more agile development processes, with

more flexibility and less documentation required. This certainly doesn't mean there is *no* process in an agile project, it just means you do what makes sense for the project you're writing so that you can satisfy all the requirements, meet the schedule, and produce a quality product.

- **The Ability to be Flexible about that Process:** No project ever proceeds as you think it will on the first day. Requirements change, people come and go, tools don't work out, and so on. This point is all about handling risk in your project. If you identify risks, plan to mitigate them, and then have a contingency plan to address the event where the risk actually occurs, you'll be in much better shape.
- **A Plan that Every One Buys Into:** You wouldn't write a sorting program without an algorithm, so you shouldn't launch a software development project without a plan. The project plan encapsulates what you're going to do to implement your project. It talks about process, risks, resources, tools, requirements management, estimates, schedules, configuration management, and delivery. It doesn't have to be long and it doesn't need to contain all the minute details of the everyday life of the project, but everyone on the team needs to have input into it, they need to understand it, and they need to agree with it. Unless everyone buys into the plan, you're doomed.
- **To Know Where you Are at All Times**: It's that communication thing again. Most projects have regular status meetings so that the developers can "sync up" on their current status and get a feel for the status of the entire project. This works very well for smaller teams (say, up to about 20 developers). Many small teams will have daily meetings to sync up at the beginning of each day. Different process models handle this "spot" meeting differently. Many plan-driven models don't require these meetings, depending on the team managers to communicate with each other. Agile processes often require daily meetings to improve communications among team members and to create a sense of camaraderie within the team.
- **To be Brave Enough to Say, "Hey, We're Behind!" :** Nearly all software projects have schedules that are too optimistic at the start. It's just the way we are. Software developers are an optimistic bunch, generally, and it shows in their estimates of work. "Sure, I can get that done in a week!" "I'll have it to you by the end of the day." "Tomorrow? Not a problem." No, no, no, no, no. Just face it. At some point you'll be behind. And the best thing to do about it is to tell your manager right away. Sure, she might be angry. But she'll be angrier when you end up a month behind and she didn't know it. Fred Brooks' famous answer to the question of how software projects get so far behind is "one day at a time." The good news, though, is that the earlier you figure out you're behind, the more options you have. These include lengthening the schedule (unlikely, but it does happen), moving some requirements to a future release, getting additional help, etc. The important part is to keep your manager informed.
- **The Right Tools and the Right Practices for this Project:** One of the best things about software development is that every project is different. Even if you're doing version 8.0 of an existing product, things change. One implication of this is that for every project one needs to examine and pick the right set of development tools for this particular project. Picking tools that are inappropriate is like trying to hammer nails with a screwdriver; you might be able to do it eventually, but is sure isn't easy or pretty, and you can drive a lot more nails in a shorter period of time with a hammer than with a screwdriver. The three most important factors in choosing tools are the application type you are writing, the target platform, and the development platform. You usually can't do anything about any of these three things, so once you know what they are, you can pick tools that improve your productivity. A fourth and nearly as important factor in tool choice is the composition and experience of the development team. If your team are all experienced developers with facility on multiple platforms tool choice is much easier. If, on the other hand, you have a bunch of fresh-outs and your target platform is new to all of you, you'll need to be careful about tool choice and fold in time for training and practice with the new tools.

To Realize that you Don't Know Everything you Need to Know at the Beginning of the

- **Project:** Software development projects just don't work this way. You'll always uncover new requirements; other requirements will be discovered to be not nearly as important as the customer thought; still others that were targeted for the next release are all of a sudden requirement number 1. Managing requirements churn during a project is one of the single most important skills a software developer can have. If you are using new development tools (say that new web development framework) you'll uncover limitations you weren't aware of and side-effects that cause you to have to learn, for example, three other tools to understand them.

1.3.2 Phases in the Development of Software

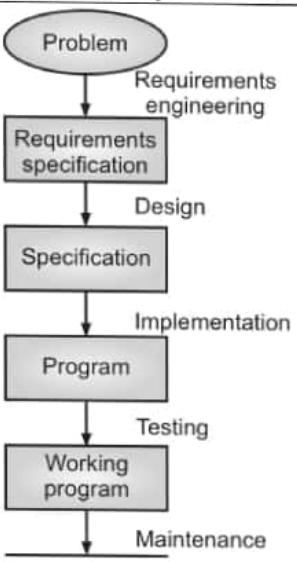


Fig. 1.3: Phases in software development

- **Requirements Engineering:** The goal of the requirements engineering phase is to get a complete description of the problem to be solved and the requirements posed by and on the environment in which the system is going to function. Requirements posed by the environment may include hardware and supporting software or the number of prospective users of the system to be developed. Alternatively, analysis of the requirements may lead to certain constraints imposed on hardware yet to be acquired or to the organization in which the system is to function. A description of the problem to be solved includes such things as:
  - ➢ The functions of the software to be developed;
  - ➢ Possible future extensions to the system;
  - ➢ The amount, and kind, of documentation required;
  - ➢ Response time and other performance requirements of the system.
- Part of requirements engineering is a **Feasibility Study**. The purpose of the feasibility study is to assess whether there is a solution to the problem which is both economically and technically feasible.
- The more careful we are during the requirements engineering phase, the larger is the chance that the ultimate system will meet expectations. To this end, the various people (among others, the customer, prospective users, designers, and programmers) involved have to collaborate intensively. The document in which the result of this activity is laid down is called the **Requirements Specification**.
- **Design:** During the design phase, a model of the whole system is developed which, when encoded in some programming language, solves the problem for the user. To this end, the problem is decomposed into manageable pieces called **components**; the functions of these components and the **interfaces** between them are specified in a very precise way. The design phase is crucial. Requirements engineering and design are sometimes seen as an annoying introduction to programming, which is often seen as the real work. This attitude has a very negative influence on the quality of the resulting software.
- Early design decisions have a major impact on the quality of the final system. These early design decisions may be captured in a global description of the system, i.e. its **architecture**. The architecture may next be evaluated, serve as a template for the development of a family of similar systems, or be used as a skeleton for the development of reusable components. As such, the architectural description of a system is an important milestone document in present-day software development projects. During the design phase we try to separate the *what* from the *how*.
- We concentrate on the problem and should not let ourselves be distracted by implementation concerns.
- The result of the design phase, the (**technical**) **specification**, serves as a starting point for the

implementation phase. If the specification is formal in nature, it can also be used to derive correctness proofs.

- **Implementation:** During the implementation phase, we concentrate on the individual components. Our starting point is the component's specification. It is often necessary to introduce an extra 'design' phase, the step from component specification to executable code often being too large. In such cases, we may take advantage of some high-level, programming-language-like notation, such as a **pseudocode**. (A pseudocode is a kind of programming language. Its syntax and semantics are in general less strict, so that algorithms can be formulated at a higher, more abstract, level.) It is important to note that the first goal of a programmer should be the development of a well-documented, reliable, easy to read, flexible, correct, program. The goal is *not* to produce a very efficient program full of tricks. During the design phase, a global structure is imposed through the introduction of components and their interfaces. In the more classic programming languages, much of this structure tends to get lost in the transition from design to code. More recent programming languages offer possibilities to retain this structure in the final code through the concept of modules or classes. The result of the implementation phase is an executable program.
- **Testing:** Actually, it is wrong to say that testing is a phase following implementation. This suggests that you need not bother about testing until implementation is finished. This is not true. It is even fair to say that this is one of the biggest mistakes you can make. Attention has to be paid to testing even during the requirements engineering phase. During the subsequent phases, testing is continued and refined. The earlier that errors are detected, the cheaper it is to correct them. Testing at phase boundaries comes in two flavors. We have to test that the transition between subsequent phases is correct (this is known as **verification**). We also have to check that we are still on the right track as regards fulfilling user requirements (**validation**).
- **Maintenance:** After delivery of the software, there are often errors that have still gone undetected. Obviously, these errors must be repaired. In addition, the actual use of the system can lead to requests for changes and enhancements. All these types of changes are denoted by the rather unfortunate term maintenance. Maintenance thus concerns all activities needed to keep the system operational after it has been delivered to the user.
- An activity spanning all phases is **project management**. Like other projects, software development projects must be managed properly in order to ensure that the product is delivered on time and within budget. The visibility and continuity characteristics of software development, as well as the fact that many software development projects are undertaken with insufficient prior experience, seriously impede project control. The many examples of software development projects that fail to meet their schedule provide ample evidence of the fact that we have by no means satisfactorily dealt with this issue yet.

1.4 SOFTWARE ENGINEERING ETHICS

- Like other engineering disciplines, software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills. You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer.
- It goes without saying that you should uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behavior are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. Confidentiality

- You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

2. Competence

- You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.

3. Intellectual Property Rights

- You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

4. Computer Misuse

- You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses or other malware).

Software Engineering Code of Ethics and Professional Practice

- Who exactly are the 'public' to whom the engineer is obligated?
- Why the software engineer is obligated to protect the public?
- What other ethical obligations software engineers are under?
- How software engineers can actually live up to ethical standards?
- What is the end goal of an ethical life in software engineering?
- What are the professional codes of software engineering ethics?

Preamble

- The Code contains eight Principles related to the behavior of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and in the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer.
- Ethical tensions can best be addressed by thoughtful consideration of fundamental principles, rather than blind reliance on detailed regulations. These Principles should influence software engineers to consider broadly who is affected by their work; to examine if they and their colleagues are treating other human beings with due respect; to consider how the public, if reasonably well informed, would view their decisions; to analyze how the least empowered will be affected by their decisions; and to consider whether their acts would be judged worthy of the ideal professional working as a software engineer. In all these judgments concern for the health, safety and welfare of the public is primary; that is, the "Public Interest" is central to this Code.
- The dynamic and demanding context of software engineering requires a code that is adaptable and relevant to new situations as they occur. However, even in this generality, the Code provides support for software engineers and managers of software engineers who need to take positive action in a specific case by documenting the ethical stance of the profession. The Code provides an ethical foundation to which individuals within teams and the team as a whole can appeal. The Code helps to define those actions that are ethically improper to request of a software engineer or teams of software engineers.

Principles

Principle 1: PUBLIC

Software engineers shall act consistently with the public interest. In particular, software engineers shall, as appropriate:

1.01. Accept full responsibility for their own work.

1.02. Moderate the interests of the software engineer, the employer, the client and the users with the public good.

1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.

1.04. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.

1.05. Cooperate in efforts to address matters of grave public concern caused by software, its installation, maintenance, support or documentation.

1.06. Be fair and avoid deception in all statements, particularly public ones, concerning software or related documents, methods and tools.

1.07. Consider issues of physical disabilities, allocation of resources, economic disadvantage and other factors that can diminish access to the benefits of software.

1.08. Be encouraged to volunteer professional skills to good causes and to contribute to public education concerning the discipline.

Principle 2: CLIENT AND EMPLOYER

Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest. In particular, software engineers shall, as appropriate:

2.01. Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.

2.02. Not knowingly use software that is obtained or retained either illegally or unethically.

2.03. Use the property of a client or employer only in ways properly authorized, and with the client's or employer's knowledge and consent.

2.04. Ensure that any document upon which they rely has been approved, when required, by someone authorized to approve it.

2.05. Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.

2.06. Identify, document, collect evidence and report to the client or the employer promptly if, in their opinion, a project is likely to fail, to prove too expensive, to violate intellectual property law, or otherwise to be problematic.

2.07. Identify, document, and report significant issues of social concern, of which they are aware, in software or related documents, to the employer or the client.

2.08. Accept no outside work detrimental to the work they perform for their primary employer.

2.09. Promote no interest adverse to their employer or client, unless a higher ethical concern is being compromised; in that case, inform the employer or another appropriate authority of the ethical concern.

Principle 3: PRODUCT

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. In particular, software engineers shall, as appropriate:

3.01. Strive for high quality, acceptable cost, and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.

3.02. Ensure proper and achievable goals and objectives for any project on which they work or propose.

3.03. Identify, define and address ethical, economic, cultural, legal and environmental issues related to work projects.

3.04. Ensure that they are qualified for any project on which they work or propose to work, by an appropriate combination of education, training, and experience.

3.05. Ensure that an appropriate method is used for any project on which they work or propose to work.

3.06. Work to follow professional standards, when available, that are most appropriate for the task at hand, departing from these only when ethically or technically justified.

3.07. Strive to fully understand the specifications for software on which they work.

3.08. Ensure that specifications for software on which they work have been well documented, satisfy the users' requirements and have the appropriate approvals.

3.09. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates.

3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work.

3.11. Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.

3.12. Work to develop software and related documents that respect the privacy of those who will be affected by that software.

3.13. Be careful to use only accurate data derived by ethical and lawful means, and use it only in ways properly authorized.

3.14. Maintain the integrity of data, being sensitive to outdated or flawed occurrences.

3.15. Treat all forms of software maintenance with the same professionalism as new development.

Principle 4: JUDGMENT

Software engineers shall maintain integrity and independence in their professional judgment. In particular, software engineers shall, as appropriate:

4.01. Temper all technical judgments by the need to support and maintain human values.

4.02. Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.

4.03. Maintain professional objectivity with respect to any software or related documents they are asked to evaluate.

4.04. Not engage in deceptive financial practices such as bribery, double billing, or other improper financial practices.

4.05. Disclose to all concerned parties those conflicts of interest that cannot reasonably be avoided or escaped.

4.06. Refuse to participate, as members or advisors, in a private, governmental or professional body concerned with software related issues, in which they, their employers or their clients have undisclosed potential conflicts of interest.

Principle 5: MANAGEMENT

Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance. In particular, those managing or leading software engineers shall, as appropriate:

5.01. Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.

5.02. Ensure that software engineers are informed of standards before being held to them.

5.03. Ensure that software engineers know the employer's policies and procedures for protecting passwords, files and information that is confidential to the employer or confidential to others.

5.04. Assign work only after taking into account appropriate contributions of education and experience tempered with a desire to further that education and experience.

5.05. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work, and provide an uncertainty assessment of these estimates.

5.06. Attract potential software engineers only by full and accurate description of the conditions of employment.

5.07. Offer fair and just remuneration.

5.08. Not unjustly prevent someone from taking a position for which that person is suitably qualified.

5.09. Ensure that there is a fair agreement concerning ownership of any software, processes, research, writing, or other intellectual property to which a software engineer has contributed.

5.10. Provide for due process in hearing charges of violation of an employer's policy or of this Code.

5.11. Not ask a software engineer to do anything inconsistent with this Code.

5.12. Not punish anyone for expressing ethical concerns about a project.

Principle 6: PROFESSION

Software engineers shall advance the integrity and reputation of the profession consistent with the public interest. In particular, software engineers shall, as appropriate:

6.01. Help develop an organizational environment favorable to acting ethically.

6.02. Promote public knowledge of software engineering.

6.03. Extend software engineering knowledge by appropriate participation in professional organizations, meetings and publications.

6.04. Support, as members of a profession, other software engineers striving to follow this Code.

6.05. Not promote their own interest at the expense of the profession, client or employer.

6.06. Obey all laws governing their work, unless, in exceptional circumstances, such compliance is inconsistent with the public interest.

6.07. Be accurate in stating the characteristics of software on which they work, avoiding not only false claims but also claims that might reasonably be supposed to be speculative, vacuous, deceptive, misleading, or doubtful.

6.08. Take responsibility for detecting, correcting, and reporting errors in software and associated documents on which they work.

6.09. Ensure that clients, employers, and supervisors know of the software engineer's commitment to this Code of ethics, and the subsequent ramifications of such commitment.

6.10. Avoid associations with businesses and organizations which are in conflict with this code.

6.11. Recognize that violations of this Code are inconsistent with being a professional software engineer.

6.12. Express concerns to the people involved when significant violations of this Code are detected unless this is impossible, counter-productive, or dangerous.

6.13. Report significant violations of this Code to appropriate authorities when it is clear that consultation with people involved in these significant violations is impossible, counter-productive or dangerous.

Principle 7: COLLEAGUES

Software engineers shall be fair to and supportive of their colleagues. In particular, software engineers shall, as appropriate:

7.01. Encourage colleagues to adhere to this Code.

7.02. Assist colleagues in professional development.

7.03. Credit fully the work of others and refrain from taking undue credit.

7.04. Review the work of others in an objective, candid, and properly-documented way.

7.05. Give a fair hearing to the opinions, concerns, or complaints of a colleague.

7.06. Assist colleagues in being fully aware of current standard work practices including policies and procedures for protecting passwords, files and other confidential information, and security measures in general.

7.07. Not unfairly intervene in the career of any colleague; however, concern for the employer, the client or public interest may compel software engineers, in good faith, to question the competence of a colleague.

7.08. In situations outside of their own areas of competence, call upon the opinions of other professionals who have competence in that area.

Principle 8: SELF

Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. In particular, software engineers shall continually endeavor to:

8.01. Further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software and related documents, together with the management of the development process.

8.02. Improve their ability to create safe, reliable, and useful quality software at reasonable cost and within a reasonable time.

8.03. Improve their ability to produce accurate, informative, and well-written documentation.

8.04. Improve their understanding of the software and related documents on which they work and of the environment in which they will be used.

8.05. Improve their knowledge of relevant standards and the law governing the software and related documents on which they work.

8.06. Improve their knowledge of this Code, its interpretation, and its application to their work.

8.07. Not give unfair treatment to anyone because of any irrelevant prejudices.

8.08. Not influence others to undertake any action that involves a breach of this Code.

8.09. Recognize that personal violations of this Code are inconsistent with being a professional software engineer.

1.5 CASE STUDIES

1. Case Study About Testing: George and the Jet

- George Babbage is an experienced software developer working for Acme Software Company. Mr. Babbage is now working on a project for the U.S. Department of Defense, testing the software used in controlling an experimental jet fighter. George is the quality control manager for the software. Early simulation testing revealed that, under certain conditions, instabilities would arise that could cause the plane to crash. The software was patched to eliminate the specific problems uncovered by the tests. After these repairs, the software passed all the simulation tests.
- George is not convinced that the software is safe. He is worried that the problems uncovered by the simulation testing were symptomatic of a design flaw that could only be eliminated by an extensive redesign of the software. He is convinced that the patch that was applied to remedy the specific tests in the simulation did not address the underlying problem. But, when George brings his concerns to his superiors, they assure him that the problem has been resolved. They further inform George that any major redesign effort would introduce unacceptable delays, resulting in costly penalties to the company.
- There is a great deal of pressure on George to sign off on the system and to allow it to be flight tested. It has even been hinted that, if he persists in delaying the system, he will be fired. What should George do next?

Relevant Clauses

Principle 1: PUBLIC

Software engineers shall act consistently with the public interest.

In particular, software engineers shall, as appropriate:

1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.

1.04. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.

Principle 3: PRODUCT

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. In particular, software engineers shall, as appropriate:

3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work.

Principle 5: MANAGEMENT

Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance. In particular, those managing or leading software engineers shall, as appropriate:

5.01. Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.

5.11. Not ask a software engineer to do anything inconsistent with this Code.

Applying the Code

- In this case, Mr. Babbage must contend with issues of physical safety that is dependent on software reliability. If we look at this case too narrowly, we might think that the safety of the test pilot is the exclusive safety concern. Although Mr. Babbage does have responsibilities towards the pilot, test pilots know about the risks inherent in their profession, and the test pilot may be quite willing to fly the plane despite Babbage's misgivings. However, the test pilot is not the only one endangered if the software is faulty; anyone under the plane is endangered if things go awry. Especially if the test flight might fly over populated areas (and remember that instability might lead the plane in unplanned directions before crashing), many people under the plane are unlikely to have given their consent to "testing" the software. Carl's responsibilities to those people are a vital part of our analysis.
- Clause 1.03 makes public safety a priority concern for a software engineer. It is exactly this concern that is central to George's decision. George clearly recognizes this obligation, and the obligation in clause 1.04 to disclose his professional opinion that the software has not been sufficiently certified as safe. Unfortunately,

George's superiors have not supported his decision about the software, and are trying to convince him to sign off on the software despite his reservations.

- His superiors have put George in a difficult position. Clearly, the Code sections above confirm Mr. Babbage's ethical duty to refuse to sign off on the software before he is reasonably sure of its safety. We note that for almost all complex software, we can never be entirely sure software is reliable and safe. It is a professional judgment whether or not the software is "safe enough." By pressuring George to sign off, his superiors are forcing George to choose between his loyalty to his employers (and his continued employment) and his obligation to public safety. It is hoped that the existence of, and support for, an effective ethics code can help someone in this position; but it is still difficult.
- So far our analysis has concentrated on Mr. Babbage and his dilemma. But the Joint Code also requires his managers to act ethically. The clauses in section 5 of the Code prohibit managers from forcing a software engineering employee to violate the code. The Code also makes managers responsible for ensuring that there are processes to ensure the reduction of risks. The managers might object that they have adequate processes, and that the process was followed. Simulation testing revealed problems, and those problems were addressed. The managers are not convinced that Mr. Babbage's suspicions are well founded, and are not willing to jeopardize the project based on his misgivings.
- The wording of clause 1.03 in the Code is an important part of our analysis of this case. That clause states that software engineers should approve software only if they have a "well-founded belief that it is safe" (our emphasis). The idea of a well-founded belief is key to the dispute between George and his superiors. Perhaps George is right about the software, but perhaps his managers are right. Although the case does not offer many details about George's misgivings, he apparently did not present sufficient evidence to his superiors about the remaining problems in the software. (If the managers were convinced about the seriousness of the remaining problems, it seems unlikely that they would approve a test flight that would likely end in a costly disaster.) Perhaps this dilemma could be resolved to the satisfaction of all parties if the managers agreed to a short term delay not for a major redesign, but for further testing to either confirm George's suspicions, or convince George that the managers are correct, and that the test flight should go on. This resolution would be far better than George signing off on a system he thinks is deficient, and far better than George being fired for not doing so. The standard supported by the Code is to have the burden to demonstrate that the software is safe before deployment instead of having to prove it unsafe before deployment is halted.

2. Case Study on Database: Levels of Security

- Leikessa Jones owns her own consulting business, and has several people working for her. Leikessa is currently designing a database management system for the personnel office of ToyTimeInc., a mid-sized company that makes toys. Leikessa has involved ToyTimeInc management in the design process from the start of the project. It is now time to decide about the kind and degree of security to build into the system.
- Leikessa has described several options to the client. The client has decided to opt for the least secure system because the system is going to cost more than was initially planned, and the least secure option is the cheapest security option. Leikessa knows that the database includes sensitive information, such as performance evaluations, medical records, and salaries. With weak security, she fears that enterprising ToyTimeInc employees will be able to easily access this sensitive data. Furthermore, she fears that the system will be an easy target for external hackers. Leikessa feels strongly that the system should be more secure than it would be if the least secure option is selected.
- Ms. Jones has tried to explain the risks to ToyTimeInc, but the CEO, the CIO, and the Director of Personnel are all convinced that the cheapest security is what they want. Should Jones refuse to build the system with the least secure option?

Relevant Clauses

Principle 1: PUBLIC

Software engineers shall act consistently with the public interest. In particular, software engineers shall, as appropriate:

1.01. Accept full responsibility for their own work.

1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.

1.04. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.

Principle 2 : CLIENT AND EMPLOYER

Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest. In particular, software engineers shall, as appropriate:

2.05. Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.

Principle 3 : PRODUCT

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. In particular, software engineers shall, as appropriate:

3.01. Strive for high quality, acceptable cost, and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.

3.03. Identify, define and address ethical, economic, cultural, legal and environmental issues related to work projects.

3.12. Work to develop software and related documents that respect the privacy of those who will be affected by that software.

Applying the Code

- Ms. Jones has competing duties to the people who hired her, the people who work at the company, to her consulting firm (including the people who work for her) and to herself. The Joint Code makes it clear that Ms. Jones must be careful about the issue of privacy; as a steward of sensitive data, she should not lose sight of that responsibility. In our first case, Carl Babbage was most concerned with avoiding physical harm to people; Ms. Jones is concerned with a different kind of harm. Both kinds are important.
- At the same time, Ms. Jones needs to balance the need for security with the economic interests of the company that hired her to do this work. Professionals have to make subjective judgments to balance cost and the customer's needs; there cannot be perfect security, and there are never infinite resources. This tension between finite resources and attaining the highest quality policy is a common cause for ethical conflicts.
- However, in this case Ms. Jones made a mistake by offering a security "option" to the company that, apparently on later reflection, she thought was inadequate. By not informing the company up front about the necessity and cost for adequate security, she has created a difficult situation, both for ToyTimeInc and for herself. In order to fulfill her obligation to the company employees, she must admit her mistake and remove that insecure system as a viable option, insisting on better security. Although the employees of ToyTimeInc haven't been consulted (at least according to this short description), they clearly will be affected by the decisions ToyTimeInc and Jones make. One possible objection to Ms. Jones not mentioning the low-security option is that she wouldn't be allowing ToyTimeInc to make an informed decision. But according to the Code, Ms. Jones is responsible for building systems that are beneficial to the public. If the low security system isn't good enough, then she shouldn't pretend that it is. An engineer designing a bridge should not be compelled to include the possibility of building it with shoddy materials in cost estimates.
- If the company refuses to upgrade the security, Ms. Jones should probably remove herself from the project if staying in the project will force her to deliver a system she thinks is unethically insecure. (Clause 1.01 is central here.) There are two objections to this suggestion. First, the company will have to find someone else to do the work, and this seems unfair to the company since they were (we assume in good faith) simply agreeing with one of Ms. Jones' suggestions. While this is unfortunate for the company

and for Ms. Jones, Ms. Jones' duty to protect sensitive information to a reasonable level of security cannot be brushed aside. A second objection is that if Ms. Jones leaves the project, the company is likely to hire someone else (who perhaps has less ethical scruples) to deliver the job with the unacceptable level of security. Although that may be true, that possible outcome does not absolve Ms. Jones of her responsibility to be an ethical professional. Ms. Jones is first and foremost responsible for her own actions; the next professional hired to take her place will have to wrestle with these responsibilities, but Ms. Jones cannot let that possibility tempt her to dodge her own responsibilities.

- There is another effect if Ms. Jones delivers the less secure system. She will have harmed the profession of software engineering by allowing a degradation of the standards for quality software. Such acts will, one software engineer at a time, reduce society's trust in software engineering as a whole.
- If ToyTimeInc insists on building the system with inadequate security, Clause 2.05 becomes important. That clause requires Ms. Jones to keep information confidential, where such confidentiality is consistent with the public interest (our emphasis). If she thinks the security is sufficiently bad, her obligation to the employees of ToyTimeInc (see clause 1.04) will take priority of the obligation for confidentiality in clause 2.05.
- Another possible solution for Jones is for her to tell ToyTimeInc that she and her consulting company will build in better security, but only charge ToyTimeInc for the cheaper option. This will hurt her financially, and may adversely affect her employees. But since Jones made the mistake of offering inadequate security is an "option," she may decide it is best for her professional reputation (and the long term success of her consulting firm) for her to absorb this loss. This option clearly fulfills her obligation to ToyTimeInc employees, although it is less clear that she has been fair to her own employees who may be harmed by the decision if her company loses money on the ToyTimeInc contract.

3. Case Study on Conflicts of Interest

- Juan Rodriguez is a private consultant who advises small businesses about their computer needs. Juan examines a company's operations, evaluates their automation needs, and recommends hardware and software to meet those needs.
- Recently, Juan was hired by a small, private hospital interested in upgrading their system for patient records and accounting. The hospital had already solicited proposals for upgrading the system, and hired Juan to evaluate the proposals they'd received.
- Juan carefully examined the proposals on the basis of the systems proposed, the experience of the companies that bid, and the costs and benefits of each proposal. He concluded that Tri-Star Systems had proposed the best system for the hospital, and he recommended that the hospital should buy the Tri-Star system. He included a detailed explanation for why he thought the Tri-Star bid was the best.
- Juan did not reveal to the hospital that he is a silent partner (a co-owner) in Tri-Star Systems. Was Juan's behavior unethical? We will assume for our discussion that Juan evaluated the bids in good faith, and sincerely believed that Tri-Star had given the best bid.

Relevant Clause

Principle 4: JUDGMENT

Software engineers shall maintain integrity and independence in their professional judgment. In particular, software engineers shall, as appropriate:

4.05. Disclose to all concerned parties those conflicts of interest that cannot reasonably be avoided or escaped.

Applying the Code

- Not all case studies require sophisticated analysis; clause 4.05 clearly labels Mr. Rodriguez's actions as wrong. Mr. Rodriguez did not fulfill his professional obligations when he neglected to disclose his conflict of interest to the hospital. Notice that his sincerity about the superiority of the Tri-Star bid is not a central issue here. The central issue is the trust Tri-Star has invested in Juan. If Mr. Rodriguez had disclosed his part ownership in Tri-Star to the hospital, and the hospital had still hired him to evaluate the bids, then Juan could have attempted to do a professional job of evaluation. (Some people might find that difficult, but it is at least theoretically possible.) However, the Code clearly prohibits Juan from taking this job without first giving the hospital the opportunity to judge for itself whether or not they wanted to hire Mr. Rodriguez despite his interest in Tri-Star.

4. Case Study : An Insulin Pump Control System

- An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.
- People who suffer from diabetes use the system. Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolizes glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.
- The problem with this treatment is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection. This can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood glucose is, in the short term, a more serious condition as it can result in temporary brain malfunctioning and, ultimately, unconsciousness and death. In the long term, however, continual high levels of blood glucose can lead to eye damage, kidney damage, and heart problems.
- Current advances in developing miniaturized sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this already exist for the treatment of hospital patients. In the future, it may be possible for many diabetics to have such systems permanently attached to their bodies.
- A software-controlled insulin delivery system might work by using a microsensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is then sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

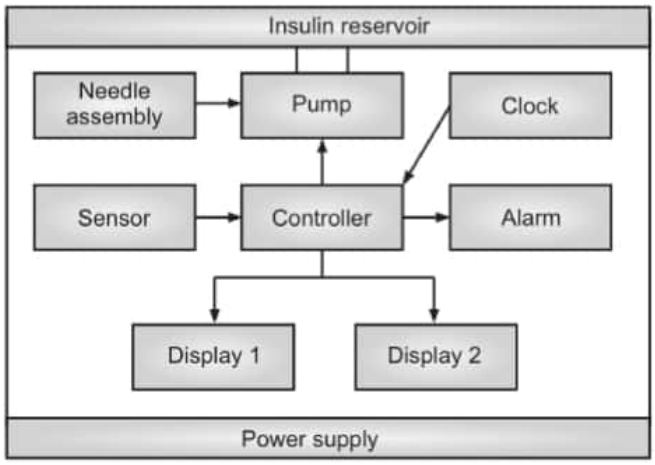


Fig. 1.4 (a) : Insulin pump hardware

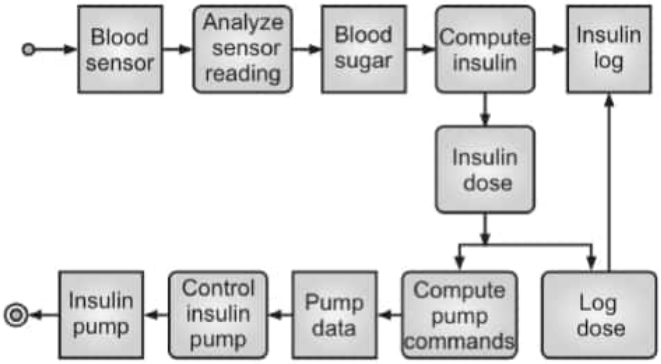


Fig. 1.4 (b) : Activity model of the insulin pump

- Fig. 1.4 (a) shows the hardware components and organization of the insulin pump. To understand the examples in this book, all you need to know is that the blood sensor measures the electrical conductivity of the blood under different conditions and that these values can be related to the blood sugar level. The insulin pump delivers one unit of insulin in response to a single pulse from a controller. Therefore, to deliver 10 units of insulin, the controller sends 10 pulses to the pump. Fig. 1.4 (b) is a UML activity model that illustrates how the software transforms an input blood sugar level to a sequence of commands that drive the insulin pump.
- Clearly, this is a safety-critical system. If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low. There are, therefore, two essential high-level requirements that this system must meet:
  - ➢ The system shall be available to deliver insulin when required.

➢ The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

5. Case Study : A Patient Information System for Mental Health Care

- A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received. Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems. To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centers.
- The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics. It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity. When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected. The system is not a complete medical records system so does not maintain information about other medical conditions. However, it may interact and exchange data with other clinical information systems. Fig. 1.5 illustrates the organization of the MHC-PMS.
- The MHC-PMS has two overall goals:
  1. To generate management information that allows health service managers to assess performance against local and government targets.
  2. To provide medical staff with timely information to support the treatment of patients.

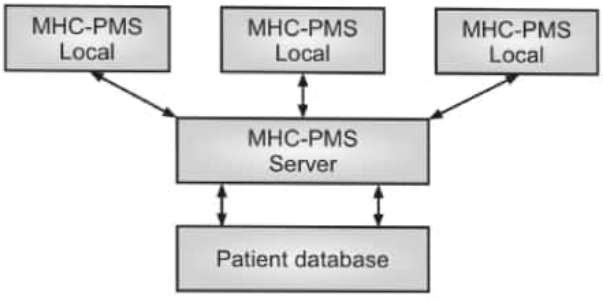


Fig. 1.5: The organization of the MHC-PMS

- The nature of mental health problems is such that patients are often disorganized so may miss appointments, deliberately or accidentally lose prescriptions and medication, forget instructions, and make unreasonable demands on medical staff. They may drop in on clinics unexpectedly. In a minority of cases, they may be a danger to themselves or to other people. They may regularly change address or may be homeless on a long-term or short-term basis. Where patients are dangerous, they may need to be 'sectioned'—confined to a secure hospital for treatment and observation.
- Users of the system include clinical staff such as doctors, nurses, and health visitors (nurses who visit people at home to check on their treatment). Nonmedical users include receptionists who make appointments, medical records staff who maintain the records system, and administrative staff who generate reports.
- The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.), conditions, and treatments. Reports are generated at regular intervals for medical staff and health authority managers. Typically, reports for medical staff focus on information about individual patients whereas management reports are anonymized and are concerned with conditions, costs of treatment, etc.

The Key Features of the System are:

1. **Individual Care Management :** Clinicians can create records for patients, edit the information in the system, view patient history, etc. The system supports data summaries so that doctors who have not previously met a patient can quickly learn about the key problems and treatments that have been prescribed.
2. **Patient Monitoring**: The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected. Therefore, if a patient has not seen a doctor for some time, a warning may be issued. One of the most important elements of the monitoring system is to keep track of patients who have been sectioned and to ensure that the legally required checks are carried out at the right time.

3. **Administrative Reporting:** The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

- Two different laws affect the system. These are laws on data protection that govern the confidentiality of personal information and mental health laws that govern the compulsory detention of patients deemed to be a danger to themselves or others. Mental health is unique in this respect as it is the only medical speciality that can recommend the detention of patients against their will. This is subject to very strict legislative safeguards. One of the aims of the MHC-PMS is to ensure that staff always act in accordance with the law and that their decisions are recorded for judicial review if necessary.
- As in all medical systems, privacy is a critical system requirement. It is essential that patient information is confidential and is never disclosed to anyone apart from authorized medical staff and the patient themselves. The MHC-PMS is also a safety-critical system. Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.
- The overall design of the system has to take into account privacy and safety requirements. The system must be available when needed otherwise safety may be compromised and it may be impossible to prescribe the correct medication to patients.
- There is a potential conflict here privacy is easiest to maintain when there is only a single copy of the system data.

6. **Case Study: A Wilderness Weather Station**

- To help monitor climate change and to improve the accuracy of weather forecasts in remote areas, the government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas. These weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed, and wind direction. Wilderness weather stations are part of a larger system (as shown in the Fig. 1.6), which is a weather information system that collects data from weather stations and makes it available to other systems for processing.
- **The Weather Station System:** This is responsible for collecting weather data, carrying out some initial data processing, and transmitting it to the data management system.
- **The Data Management and Archiving System:** This system collects the data from all of the wilderness weather stations, carries out data processing and analysis, and archives the data in a form that can be retrieved by other systems, such as weather forecasting systems.
- **The Station Maintenance System:** This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems. It can update the embedded software in these systems. In the event of system problems, this system can also be used to remotely control a wilderness weather system.

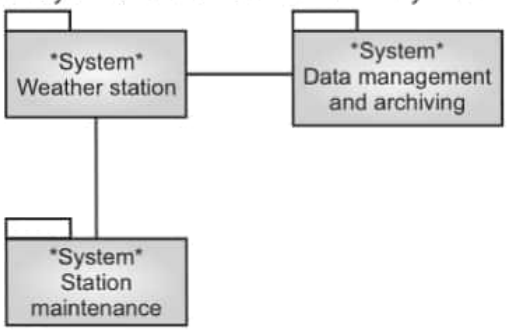


Fig. 1.6 : The weather station's environment

- In this Fig. 1.6, UML package symbols are used to indicate that each system is a collection of components and have identified the separate systems, using the UML stereotype «system». The associations between the packages indicate there is an exchange of information but, at this stage, there is no need to define them in any more detail.
- Each weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure, and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

- The weather station system operates by collecting weather observations at frequent intervals for example, temperatures are measured every minute. However, because the bandwidth to the satellite is relatively narrow, the weather station carries out some local processing and aggregation of the data. It then transmits this aggregated data when requested by the data collection system. If, for whatever reason, it is impossible to make a connection, then the weather station maintains the data locally until communication can be resumed.
- Each weather station is battery-powered and must be entirely self-contained there are no external power or network cables available. All communications are through a relatively slow-speed satellite link and the weather station must include some mechanism (solar or wind power) to charge its batteries. As they are deployed in wilderness areas, they are exposed to severe environmental conditions and may be damaged by animals.
- The station software is therefore not just concerned with data collection. It must also:
  - ➢ Monitor the instruments, power, and communication hardware and report faults to the management system.
  - ➢ Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
  - ➢ Allow for dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.
- Because weather stations have to be self-contained and unattended, this means that the software installed is complex, even though the data collection functionality is fairly simple.

7. Case Study : A Digital Learning Environment for Schools

- A digital learning environment is a framework in which a set of general-purpose and specially designed tools for learning may be embedded, plus a set of applications that are geared to the needs of the learners using the system. The framework provides general services such as an authentication service, synchronous and asynchronous communication services, and a storage service.
- The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs. These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games, and simulations.
- Fig. 1.7 shows a high-level architectural model of a digital learning environment (iLearn) that was designed for use in schools for students from 3 to 18 years of age. The approach adopted is that this is a distributed system in which all components of the environment are services that can be accessed from anywhere on the Internet. There is no requirement that all of the learning tools are gathered together in one place.
- The system is a service-oriented system with all system components considered to be a replaceable service. There are three types of service in the system:
  - **(i) Utility Services** that provide basic application-independent functionality and that may be used by other services in the system. Utility services are usually developed or adapted specifically for this system.
  - **(ii) Application Services** that provide specific applications such as email, conferencing, photo sharing, etc., and access to specific educational content such as scientific films or historical resources. Application services are external services that are either specifically purchased for the system or are available freely over the Internet.
  - **(iii) Configuration Services** that are used to adapt the environment with a specific set of application services and to define how services are shared between students, teachers, and their parents.

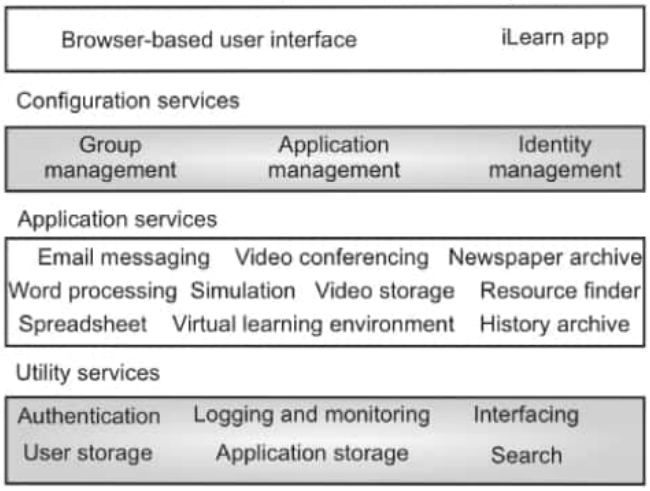


Fig. 1.7: The architecture of a digital learning environment (iLearn)

- The environment has been designed so that services can be replaced as new services become available and to provide different versions of the system that are suited for the age of the users. This means that the system has to support two levels of service integration:
  - (i) **Integrated Services are Services** that offer an API (application programming interface) and that can be accessed by other services through that API. Direct service-to-service communication is therefore possible. An authentication service is an example of an integrated service. Rather than use their own authentication mechanisms, an authentication service may be called on by other services to authenticate users. If users are already authenticated, then the authentication service may pass authentication information directly to another service, via an API, with no need for users to reauthenticate themselves.
  - (ii) **Independent Services are Services** that are simply accessed through a browser interface and that operate independently of other services. Information can only be shared with other services through explicit user actions such as copy and paste; reauthentication may be required for each independent service.
- If an independent service becomes widely used, the development team may then integrate that service so that it becomes an integrated and supported service.

1.6 PROCESS MODELS

1.6.1 Generic Process Model

Software Process Framework

There are five generic Process Framework activities :

1. **Communication :** Communication tasks required to establish effective communication between developer and customer to get requirements.
2. **Planning :** Planning tasks required to define resources, timelines, and other project related information.
3. **Modeling :** Modeling tasks required to do detail requirement analysis and project design (algorithm, flowchart, etc).
4. **Construction :** It includes coding and testing.
   - **Coding :** The design must be translated into a machine-readable form. The code generation step performs this task.
   - **Testing :** The testing process focuses on:
     - ➢ The logical internals of the software, ensuring that all statements have been tested.
     - ➢ The functional externals; that are, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.
5. **Deployment :** It includes software delivery, support and feedback from customer. If customer want some changes, correction, additional functionality that has to be done. A software process can be characterized as shown in Fig. 1.8. A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of task sets (each a collection of software engineering work tasks, project milestones, work products, and quality assurance points) enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Umbrella activities overlay the process model which are applied throughout process.

Software Project Tracking and Control

- The purpose of project tracking and control is to establish an overall approval for monitoring the quality of tasks, schedule, and budget performance to plan and taking effective corrective actions. Project control is the process of project management reviews by comparing actual performance with planned performance; analyzing variances; evaluating possible alternatives; and taking appropriate, immediate, and effective corrective or continuative action as needed.

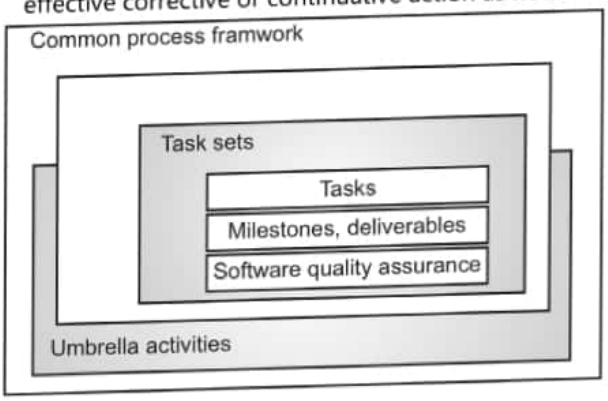


Fig. 1.8 : Software Process Framework

Risk Management

- It is the identification, assessment, and prioritization of risks to minimize, monitor, and control the effect of unfortunate events or to maximize the realization of opportunities.

Software Quality Assurance

- Software Quality Assurance (SQA) is a process that ensures that developed software meets and complies with defined or standardized quality specifications. SQA is an ongoing process within the software development life cycle (SDLC) that routinely checks the developed software to ensure it meets desired quality measures.

Formal Technical Reviews (FTR)

- FTR is a meeting conducted by technical staff. The objectives of the FTR are :
  - ➢ To uncover errors in function, logic, or implementation for any representation of the software;
  - ➢ To verify that the software under review meets its requirements;
  - ➢ To ensure that the software has been represented according to predefined standards;
  - ➢ To achieve software that is developed in a uniform manner;
  - ➢ To make projects more manageable.
- In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation.

Measurement

- Measurement and analysis involves gathering quantitative data about products, processes, and projects and analyzing that data to influence actions and plans. Measurement and analysis activities allow to:
  - ➢ Characterize, or gain understanding of processes, products, resources, and environments.
  - ➢ Evaluate, to determine status with respect to plans.
  - ➢ Predict, by understanding relationships among processes and products so the values observe for some attributes can be used to predict others.
  - ➢ Improve, by identifying roadblocks, root causes, inefficiencies, and other opportunities for improvement.

Software Configuration Management (SCM)

- SCM helps in identifying individual elements and configurations, tracking changes, and version selection, control, and base-lining of software system.

Reusability Management

- Software reuse is the process of implementing or updating software systems using existing software components. A good software reuse process facilitates the increase of productivity, quality, and reliability, and the decrease of costs and implementation time.

Work Product Preparation and Production

- In includes the activities like documentation, log formation, provide manuals for developed software.

1.6.2 Prescriptive Process Models

1. The Waterfall Model

- Waterfall model is the oldest and has an important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing and support can be placed.

Communication

- Software development process starts with communication between customer and developer. Customer must state all the requirements at the beginning of project to understand the nature of software to be built. Developer must understand the information domain.

Planning

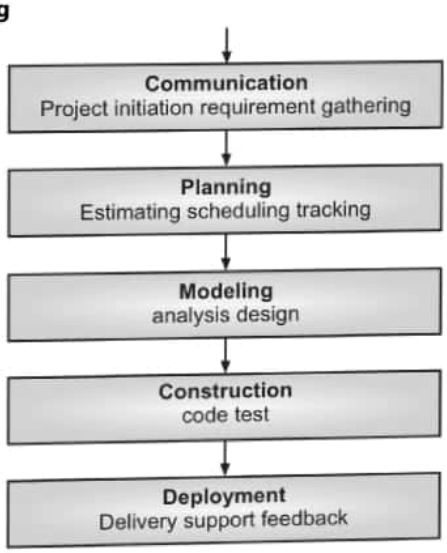


Fig. 1.9 : Waterfall Model

- It includes task required defining resources, timelines, cost estimation and other project related information. It is necessary to keep track of project whether it is going as per plan or not which is nothing but tracking.

Modeling

- Modeling is necessary for system visualization and understanding of system structure and behavior (use of algorithms, flowchart etc). Flowchart shows pictorial flow of program. Modeling includes analysis and design. Software design is a multistep process that focuses on four distinct attributes of the program.
  1. Data structure.
  2. Software architecture.
  3. Procedural detail, and
  4. Interface characterization.

Construction

- It includes coding and testing phase:
  - ➢ **Coding :** The code generation step performs translation of design details into a machine readable form.
  - ➢ **Testing :** Testing is conducted to uncover the errors and to ensure that defined input will produce the result that agree with required result.

Deployment

- It includes handover of project to customer, taking feedback from customer and provides future support.

Advantages of Waterfall Model

- Waterfall model is a systematic sequential approach for software development. Hence it is most widely used paradigm for software development.
- It gives straightforward way to divide the large task into series of cleanly separated phases.

Disadvantages of Waterfall Model

- Problems in this model remain uncovered until software testing.
- It is often difficult for the customer to state all requirements explicitly.
- The customer must have patience; working version of the program(s) will not be available until late in the project time-span.

When to Use the Waterfall Model

- This model is used only when the requirements are very well known, clear and fixed.
- Product definition is stable.
- Technology is understood.
- There are no ambiguous requirements
- Ample resources with required expertise are available freely
- The project is short.

2. Incremental Process Models

- The incremental process model is an evolution of the waterfall model, it combines element of waterfall model (applied repeatedly) with the iterative philosophy of prototyping.

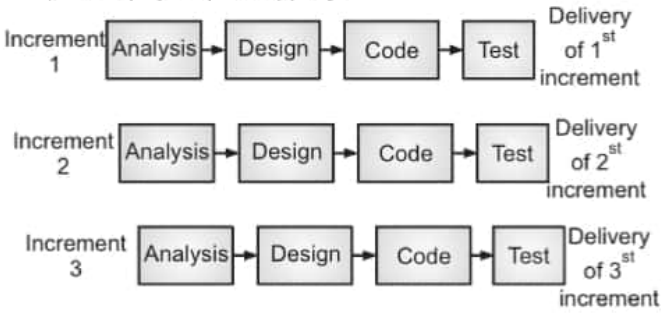


Fig. 1.10 : Incremental Model

- The series of releases as "increments", with each increment providing more functionality to the customers. After the first increment a core product is delivered. Based on customer feedback, a plan is developed for the next increments, and modifications are made accordingly. This process continues with increment being delivered until the complete product is delivered.
- For Example. Word processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.
- When the incremental model is used, the first increment is often a core product, that is only basic requirements are addressed. Core product is reviewed to plan next increment.

Advantages of Incremental Process Model

- Enables partial functionality to be delivered to end users without inordinate delay.
- Can be used when sufficient staff is not available.

- After each iteration, regression testing should be conducted. During this testing, faulty element of the software can be quickly identified.
- It is generally easier to test and debug then other methods of software developed because relatively smaller changes are made during each iteration.

Disadvantages of Incremental Process Model

- Resulting cost may exceed the cost of the organization.
- As additional functionality is added to the product, problems may arise related to system architecture when were not evident in earlier prototypes.

When to use the Incremental Model:

- This model can be used when the requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some details can evolve with time.
- There is a need to get a product to the market early.
- A new technology is being used
- Resources with needed skill set are not available
- There are some high risk features and goals.

3. Rapid Application Development (RAD) Model

- Rapid Application Development is a linear sequential software development model that emphasizes an extremely short development.
- Used primarily for information systems applications, the RAD approach encompasses the following phases, illustrated in Fig. 1.11.
  - ➢ Business modeling (BM)
  - ➢ Process Modeling (PM)
  - ➢ Testing and Turnover (TT)
  - ➢ Data modeling (DM)
  - ➢ Application Generation (AG)

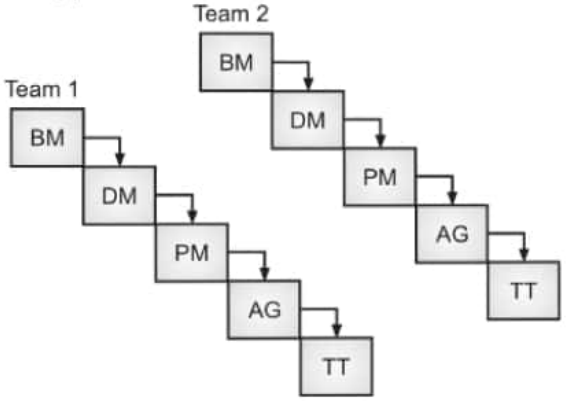


Fig. 1.11 : RAD Model

- The information flow among business functions is modeled in a way that answers the following questions.
  - ➢ What information drives that business process?
  - ➢ What information is generated?
  - ➢ Who generates it?
  - ➢ Where does the information go?
  - ➢ Who processes it?
- The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The attributes/characteristics of each object are identified and the relationships between these objects are defined.
- Processing descriptions are created for adding, modifying deleting or retrieving a data object.
- RAD process works to reuse existing program components or create reusable components. In all cases, automated tools are used to facilitate construction of the software.
- Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.
- Like all process models, the RAD approach has drawbacks such as: for large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to the rapid-fire activities necessary to complete a system in much abbreviated time frame. If commitment is lacking form either constituency, RAD will fail.

When to use RAD Model:

- RAD should be used when there is a need to create a system that can be modularized in 2-3 months of time.
- It should be used if there's high availability of designers for modeling and the budget is high enough to afford their cost along with the cost of automated code generating tools.
- RAD SDLC model should be chosen only if resources with high business knowledge are available and there is a need to produce the system in a short span of time (2-3 months).

1.6.3 Evolutionary Process Models

1. Prototype Model

- The basic idea in **Prototype model** is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Prototype model is a software development model. By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system. Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.
- The prototype are usually not complete systems and many of the details are not built in the prototype. The goal is to provide a system with overall functionality.

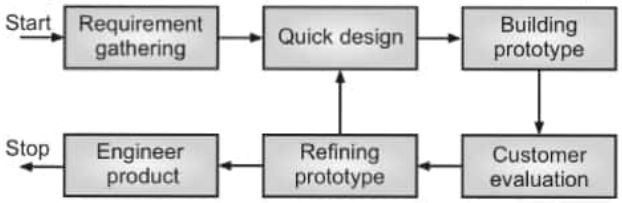


Fig. 1.12 : Prototype model

Advantages of Prototype Model:

- Users are actively involved in the development
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily
- Confusing or difficult functions can be identified Requirements validation, Quick implementation of, incomplete, but functional, application.

Disadvantages of Prototype Model:

- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Incomplete application may cause application not to be used as the full system was designed incomplete or inadequate problem analysis.

When to use Prototype Model:

- Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
- Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
- Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

2. Spiral Model

- Spiral model combines the iterative nature of prototyping with controlled and systematic aspect of the linear sequential model. Using the spiral model, software is developed in a series of incremental release.

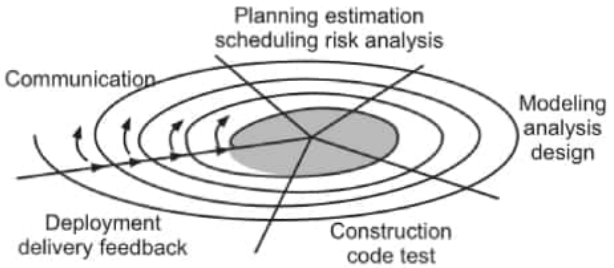


Fig. 1.13 : Waterfall Model

- Spiral model can be applied throughout the software development life cycle. During early iteration, increment release might be a paper model or prototype. During later iteration more detailed version of software may be produced. Spiral model is divided into number of framework activities (also called task region). In this model software engineering team begins at the center and moves around the spiral in a clockwise direction.

(i) Communication

Task required for establishing effective communication between developer and customer.

(ii) Planning

Task required to define resources, timelines and other project related information.

(iii) Modeling

It includes detail requirement analysis and project design (algorithm and flowchart).

(iv) Construction

Design details are translated into machine readable form and testing is conducted to uncover the errors.

(v) Deployment

It includes software delivery, support and feedback from customer.

Example: Documentation and training.

Advantages of Spiral Model

- Spiral model is a realistic approach to the development of large scale systems and software.
- The spiral model demands direct consideration of technical risks at all stages of the project and if properly applied, can reduce risk before they become problematic.

Disadvantages of Spiral Model

- It may be difficult to convince customer that evolutionary approach is controllable.
- If major risk is not uncovered and managed at beginning problems may occur.

When to use Spiral Model:

- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

3. Concurrent Models

- The concurrent development model sometimes called concurrent engineering, provides an accurate view of the current state of a project. It focuses on concurrent engineering activities in a software engineering process such as prototyping, modeling, requirement specification and design. The concurrent model is often more appropriate for system engineering project where different engineering teams are involved. All activities exist concurrently but reside in different states.

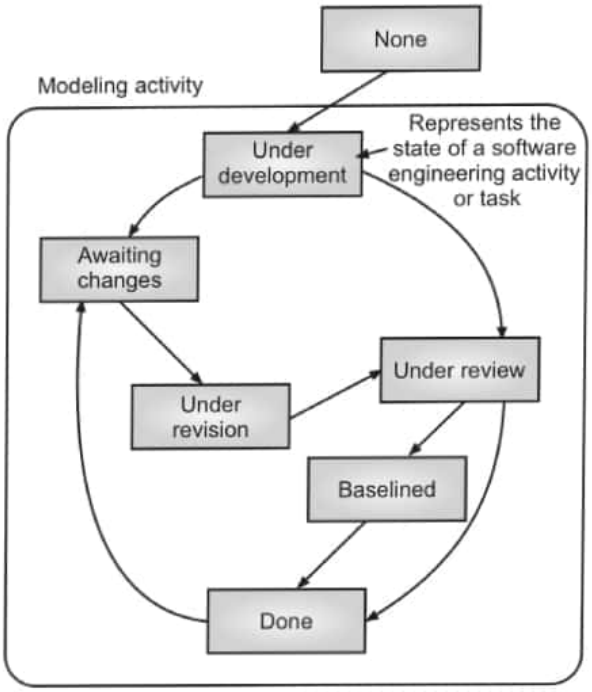


Fig. 1.14 : Concurrent Models

- For Example. early in the project the communication activity has completed its first iteration and exist in the 'awaiting changes' state. The modeling activity which existed in the 'none' state will transit to 'under development' state on completion of initial communication. If, however, the customer indicates the changes in requirement must be made, the modeling activity moves from the under development state into the 'awaiting changes' state.
- Concurrent process model is applicable to all types of software development. It defines network of software engineering activities. Changes in one activity can trigger transition among states of an activity.

Advantages of the Concurrent Development Model

- This model is applicable to all types of software development processes.
- It is easy for understanding and use.
- It gives immediate feedback from testing.
- It provides an accurate picture of the current state of a project.

Disadvantages of the Concurrent Development Model

- It needs better communication between the team members. This may not be achieved all the time.
- It requires to remember the status of the different activities.

1.6.4 V Model

- V- model means Verification and Validation model. Just like the **waterfall model**, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. **V-Model** is one of the **many software development models**.
- Testing of the product is planned in parallel with a corresponding phase of development in **V-model**.

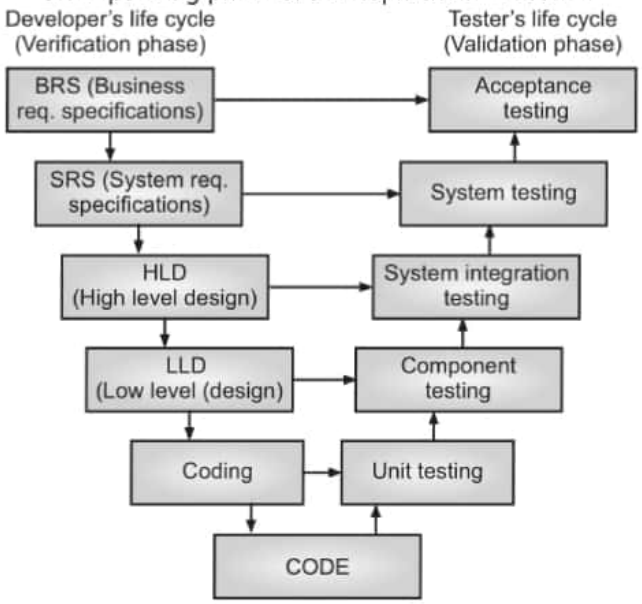


Fig. 1.15 : V-model

The various phases of the V-model are as follows:

- **Requirements** like BRS and SRS begin the life cycle model just like the waterfall model. But, in this model before development is started, a **system test** plan is created. The **test plan** focuses on meeting the functionality specified in the requirements gathering.
- **The High-Level Design (HLD)** phase focuses on system architecture and design. It provides overview of solution, platform, system, product and service/process. An **integration test** plan is created in this phase as well in order to test the pieces of the software systems ability to work together.
- **The Low-Level Design (LLD)** phase is where the actual software components are designed. It defines the actual logic for each and every component of the system. Class diagram with all the methods and relation between classes comes under LLD. **Component tests** are created in this phase as well.
- **The Implementation** phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.
- **Coding:** This is at the bottom of the V-Shape model. Module design is converted into code by developers. **Unit Testing** is performed by the developers on the code written by them.

Advantages of V-Model:

- Simple and easy to use.
- Testing activities like planning, **test designing** happens well before coding. This saves a lot of time. Hence higher chance of success over the waterfall model.
- Proactive defect tracking – that is defects are found at early stage.
- Avoids the downward flow of the defects.
- Works well for small projects where requirements are easily understood.

Disadvantages of V-Model:

- Very rigid and least flexible.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- If any changes happen in midway, then the test documents along with requirement documents has to be updated.

When to use the V-Model:

- The V-shaped model should be used for small to medium sized projects where requirements are clearly defined and fixed.
- The V-Shaped model should be chosen when ample technical resources are available with needed technical expertise.
- High confidence of customer is required for choosing the V-Shaped model approach. Since, no prototypes are produced, there is a very high risk involved in meeting customer expectations.

1.6.5 Iterative Model

- An iterative life cycle model does not attempt to start with a full specification of requirements by first focusing on an initial, simplified set user features, which then progressively gains more complexity and a broader set of features until the targeted system is complete. When adopting the iterative approach, the philosophy of incremental development will also often be used liberally and interchangeably.
- In other words, the iterative approach begins by specifying and implementing just part of the software, which can then be reviewed and prioritized in order to identify further requirements. This iterative process is then repeated by delivering a new version of the software for each iteration. In a light-weight iterative project the code may represent the major source of documentation of the system; however, in a critical iterative project a formal software specification may also be required.

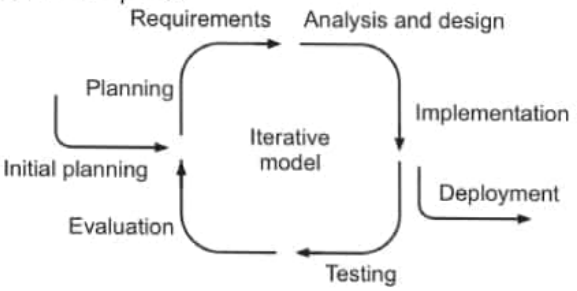


Fig. 1.16 : Iterative model

When to use Iterative Model:

- Requirements of the complete system are clearly defined and understood.
- When the project is big.
- Major requirements must be defined; however, some details can evolve with time.

1.6.6 Reuse-Oriented Software Engineering

- Today some software is reuse able, we use basic architecture of system such as design, code etc and changes are made if required and incorporate into system. When people working on software project if design and code are similar to other software then we use that software and modify them according to our need and include them into their system. Today the use of existing software widely spread. Reusable software models save development time of the project.
- Reuse –oriented software base on reusable components and integrated framework for the composition of these components, that components may provide specific function such as word processing and spreadsheet. Type of software components that is used in reuse-oriented software process are
- Web services, Services standard are used for development these standard are available for remote. Objects integrated with component framework, these object is created as a package. Stand alone software system can be configured. That software is used in particular environment.

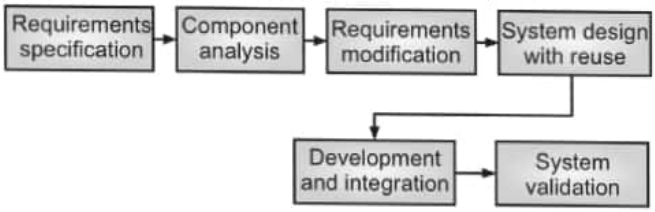


Fig. 1.17 : Reuse-oriented software engineering

- General process of reuse-oriented model are shown in Fig. 1.17.
- Requirement specification and system validation stages are general process used in different software process but other stages used in this model are different.

1. **Component Analysis:**

- According to given requirement, component is selected to implement that requirement specification. That is not possible the selected component provide the complete functionality, but that is possible the component used provide some of the functionality required.

2. **Requirement Modification:**

- Information about component that is selected during component analysis is used to analysis requirement specification. Requirements are modified according to available components. Requirement modification is critical then component analysis activity is reused to find relative solution.

3. **System Design with Reuse:**

- During this stage the design of the system is build. Designer must consider the reused component and organize the framework. If reused component is not available then new software is develop.

4. Development and Integration:

- Components and COTS system are integrated to develop new software. Integration in this model is part of development rather than separate activity.

Advantages of Reuse-Oriented Model:

- It can reduce the overall cost of software development as compared to other model.
- It can reduce the risk.
- It can save the time of software development. b/c testing of component is minimize.
- Faster delivery of software.

Disadvantages of Reuse-Oriented Model:

- Reuse-oriented model is not always practical in its pure form.
- Compromises in Requirement may lead to a system that does not meet the real requirement of the user.
- Organization using the reusable component, are not able to control the new version of component, this may lead to lost control over the system evolution.

1.7 PROCESS ACTIVITIES

- The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. There are four fundamental activities that are common to all software processes. These activities are:
  1. **Software Specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
  2. **Software Design and Implementation**, where the software is designed and programmed.
  3. **Software Validation**, where the software is checked to ensure that it is what the customer requires.
  4. **Software Evolution**, where the software is modified to reflect changing customer and market requirements.
- The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in evolutionary development they are interleaved. How these activities are carried out depends on the type of software, people and organizational structures involved.

1. Software Specification

- A software requirement is defined as a condition to which a system must comply. Software specification or requirements management is the process of understanding and defining what functional and non-functional requirements are required for the system and identifying the constraints on the system's operation and development. The requirements engineering process results in the production of a software requirements document that is the specification for the system.
- There are four main phases in the requirements engineering process:
  - **(i) Feasibility Study:** In this study an estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and whether it can be developed within existing budgetary constraints.
  - **(ii) Requirements Elicitation and Analysis:** This is the process of deriving the system requirements through observation of existing systems, discussions with potential users, requirements workshop, storyboarding, etc.
  - **(iii) Requirements Specification:** This is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document: user (functional) requirements and system (non-functional) requirements.
  - **(iv) Requirements Validation:** It is determined whether the requirements defined are complete. This activity also checks the requirements for consistency.

2. Software Design and Implementation

- The implementation phase of software development is the process of converting a system specification into an executable system through the design of system. A software design is a description of the architecture of the software to be implemented, the data which is part of the system, the interfaces between system components and, sometimes, the algorithms used. The design process activities are the followings:
  - ➢ **Architectural Design:** The sub-systems of system and their relationships are identified based on the main functional requirements of software.
  - ➢ **Abstract Specification:** For each sub-system, an abstract specification of its services and the constraints under which it must operate is defined.
  - ➢ **Interface Design:** Interfaces allow the sub-system' services to be used by other sub-systems. The representation of interface should be hidden. In this activity the interface is designed and documented for each sub-system. The specification of interface must be unambiguous.
  - ➢ **Component Design:** Services are allocated to components and the interfaces of these components are designed.
  - ➢ **Data Structure Design:** The data structures used in the system implementation are designed in detail and specified.
  - ➢ **Algorithm Design:** In this activity the algorithms used to provide services are designed in detail and specified.
- This general model of the design process may be adapted in different ways in the practical uses.
- A contrasting approach can be used by structured methods for design objectives. A structured method includes a design process model, notations to represent the design, report formats, rules and design guidelines. Most these methods represent the system by graphical models and many cases can automatically generate program code from these models. Various competing methods to support object-oriented design were proposed in the 1990s and these were unified to create the Unified Modeling Language (UML) and the associated unified design process.

3. Software Validation

- Software validation or, more generally, Verification and Validation (V & V) is intended to show that a system conforms to its specification and that the system meets the expectations of the customer buying the system. It involves checking the processes at each stage of the software process. The majority of validation costs are incurred after implementation when the operation of system is tested.
- The software is tested in the usual three-stage testing process. The system components, the integrated system and finally the entire system are tested. Component defects are generally discovered early in the process and the interface problems during the system integration. The stages in the testing process are:
  - ➢ **Component (or Unit) Testing:** Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components.
  - ➢ **System Testing:** The components are integrated to make up the system. This testing process is concerned with finding errors that result from interactions between components and component interface problems. It is also concerned with validating that the system meets its functional and non-functional requirements.
  - ➢ **Acceptance Testing:** It is considered a functional testing of system. The system is tested with data supplied by the system customer.
- Usually, component development and testing are interleaved. Programmers make up their own test data and test the code as it is developed. However in many process model, such as in V-model, Test Driven Development, Extreme Programming, etc., the design of the test cases starts before the implementation phase of development. If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment.

4. Software Evolution

- Software evolution, specifically software maintenance, is the term used in software engineering to refer to the process of developing software initially, then repeatedly updating it for various reasons.
- The aim of software evolution would be to implement the possible major changes to the system. The existing larger system is never complete and continues to evolve. As it evolves, the complexity of the system will grow. The main objectives of software evolution are ensuring the reliability and flexibility of the system. The costs of maintenance are often several times the initial development costs of software.

1.8 COPING WITH CHANGE

- Coping with change Coping with change or sometime called "Process iteration" means that, change is sure to happen in all large software projects. As new technologies become available, new design and implementation possibilities emerge. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed. Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and re-test the system.
- There are two related approaches that may be used to reduce the costs of rework:

1. **Change Avoidance**, where the software process includes activities that can anticipate possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.
2. **Change Tolerance**, where the process is designed so that changes can be accommodated at relatively low cost. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed.

- There are two ways of coping with change and changing system requirements. These are:

(i) **System Prototyping**, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions. This supports change avoidance as it allows users to experiment with the system before delivery and so refine their requirements. The number of requirements change proposals made after delivery is therefore likely to be reduced.

(ii) **Incremental Delivery**, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

1. Software Prototyping

- A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- A prototype can be used in:
  - ➢ The requirements engineering process to help with requirements elicitation and validation;
  - ➢ In design processes to explore options and develop a UI design;
  - ➢ In the testing process to run back-to-back tests.

Benefits of Prototyping

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

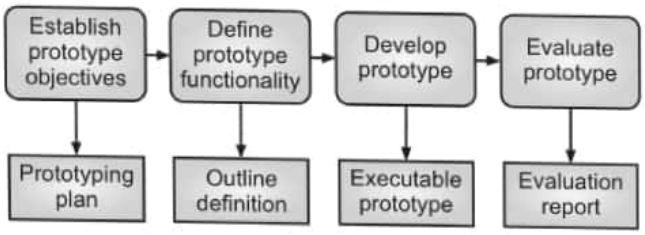


Fig. 1.18 : The process of prototype development

Prototype Development

- May be based on rapid prototyping languages or tools
- May involve leaving out functionality
- Prototype should focus on areas of the product that are not well-understood;
- Error checking and recovery may not be included in the prototype;
- Focus on functional rather than non-functional requirements such as reliability and security

Throw-Away Prototypes

- Prototypes should be discarded after development as they are not a good basis for a production system:
  - ➢ It may be impossible to tune the system to meet non-functional requirements;
  - ➢ Prototypes are normally undocumented;
  - ➢ The prototype structure is usually degraded through rapid change;
  - ➢ The prototype probably will not meet normal organisational quality standards.

2. Incremental Delivery

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

Incremental Development and Delivery

- Incremental development
  - ➢ Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
  - ➢ Normal approach used in agile methods;
  - ➢ Evaluation done by user/customer proxy.
- Incremental delivery
  - ➢ Deploy an increment for use by end-users;
  - ➢ More realistic evaluation about practical use of software;
  - ➢ Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

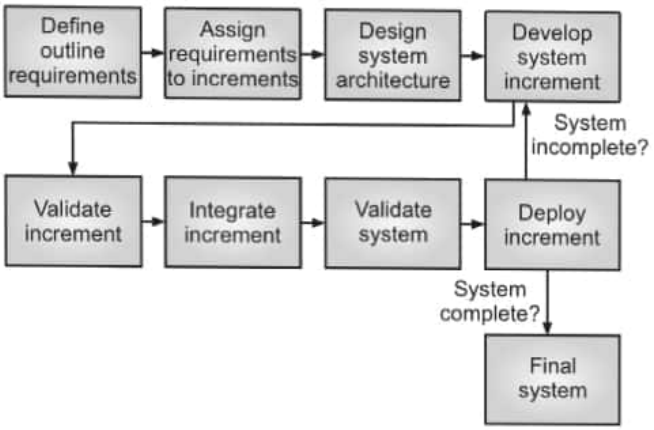


Fig. 1.19 : Incremental delivery

Incremental Delivery Advantages

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

Incremental Delivery Problems

- Most systems require a set of basic facilities that are used by different parts of the system.
  - ➢ As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.

- The essence of iterative processes is that the specification is developed in conjunction with the software.
  - ➢ However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

3. Boehm's Spiral Model

- A risk-driven software process framework (the spiral model) was proposed by Boehm (1988). This is shown in next Fig. 1.20. Rather than represent the software process as a sequence of activities with some backtracking from one activity to another, the process is represented as a spiral .Each loop in the spiral represents a phase of the software process. Thus, the deepest loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model 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.

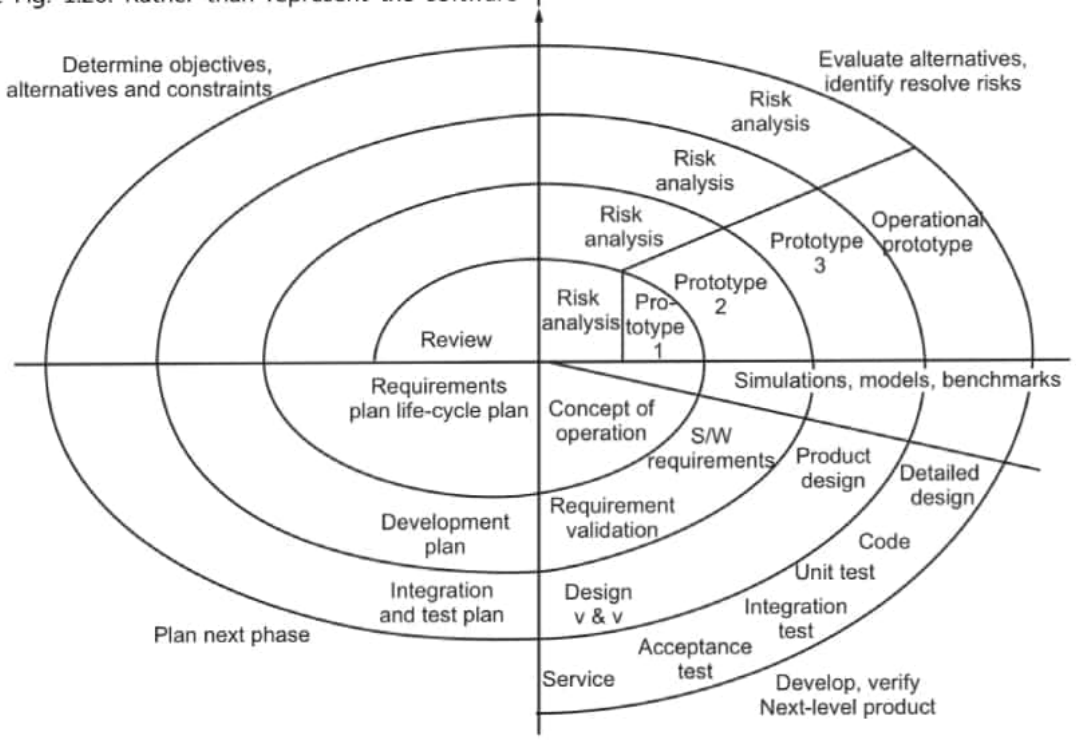


Fig. 1.20

Each loop in the spiral is split into four sectors:

1. Objective Setting

- Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.

2. Risk Assessment and Reduction

- For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. Development and Validation

- After risk evaluation, a development model for the system is chosen. For example, throwaway prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system

integration, the waterfall model may be the best development model to use.

4. Planning

- The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.
- The main difference between the spiral model and other software process models is its explicit recognition of risk. A cycle of the spiral begins by elaborating objectives such as performance and functionality. Alternative ways of achieving these objectives and dealing with the constraints on each of them are then enumerated. Each alternative is assessed against each objective and sources of project risk are identified. The next step is to resolve these risks by information-gathering activities such as more detailed analysis, prototyping, and simulation.
- Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code. Risks lead to proposed software changes and project problems such as schedule and cost overrun, so risk minimization is a very important project management activity.

1.9 THE RATIONAL UNIFIED PROCESS

- The Rational Unified Process (RUP) methodology is an example of a modern software process model that has been derived from the UML and the associated Unified Software Development Process. The RUP recognizes that conventional process models present a single view of the process. In contrast, the RUP is described from three perspectives:
  1. A dynamic perspective that shows the phases of the model over time.
  2. A static perspective that shows the process activities.
  3. A practice perspective that suggests good practices to be used during the process.

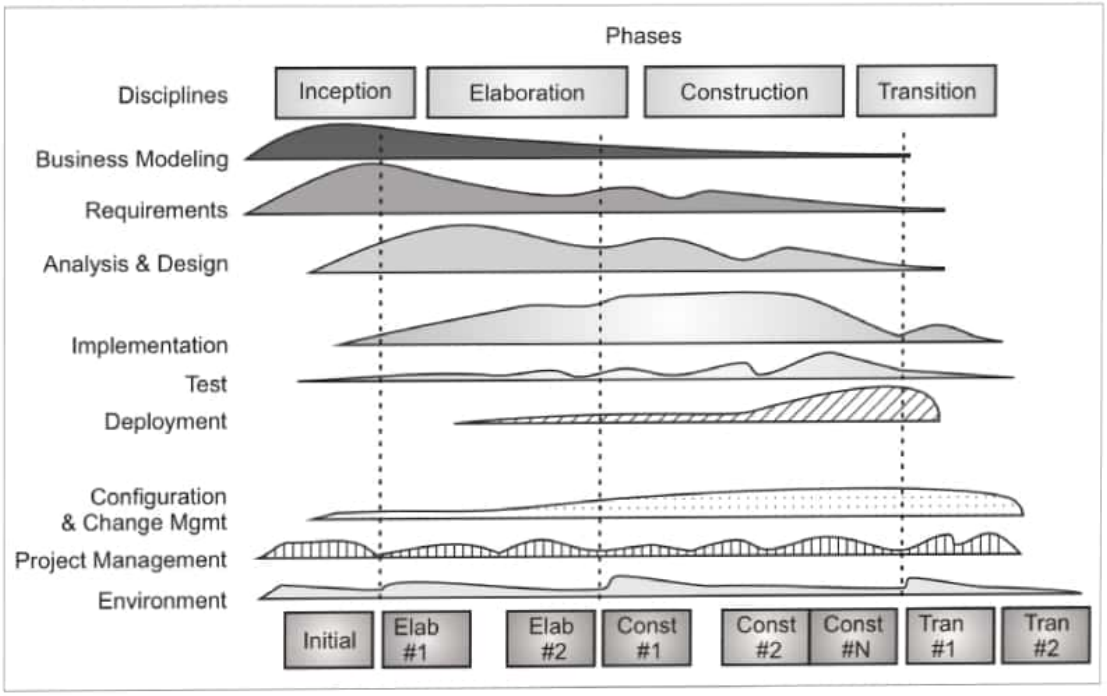


Fig. 1.21 : Phases of Rational Unified Process

- The RUP identifies four discrete development phases in the software process that are not equated with process activities. The phases in the RUP are more closely related to business rather than technical concerns. These phases in the RUP are:
  - ➢ **Inception:**. The goal of the inception phase is to establish a business case for the system, identify all external entities, i.e. people and systems that will interact with the system and define these interactions. Then this information is used to assess the contribution that the system makes to the business.
  - ➢ **Elaboration:** The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan and identify key project risks.
  - ➢ **Construction**: The construction phase is essentially concerned with system design, programming and testing.
  - ➢ **Transition**: The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment.
- Iteration within the RUP is supported in two ways. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally.
- The static view of the RUP focuses on the activities that take place during the development process. These are called workflows in the RUP description. There are six core process workflows identified in the process and three core supporting workflows. The RUP has been designed in conjunction with the UML so the workflow description is oriented around associated UML models. The core engineering and support workflows are the followings:
  - ➢ **Business Modeling**: The business processes are modelled using business use cases.
  - ➢ **Requirements**: Actors who interact with the system are identified and use cases are developed to model the system requirements.
  - ➢ **Analysis and Design:** A design model is created and documented using architectural models, component models, object models and sequence models.
  - ➢ **Implementation:** The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
  - ➢ **Testing:** Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
  - ➢ **Deployment:** A product release is created, distributed to users and installed in their workplace.
  - ➢ **Configuration and Change Management:** This supporting workflow manages changes to the system.
  - ➢ **Project Management**: This supporting workflow manages the system development.
  - ➢ **Environment:** This workflow is concerned with making appropriate software tools available to the software development team.
- The advantage in presenting dynamic and static views is that phases of the development process are not associated with specific workflows. In principle at least, all of the RUP workflows may be active at all stages of the process. Of course, most effort will probably be spent on workflows such as business modelling and

requirements at the early phases of the process and in testing and deployment in the later phases.

- The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development. Six fundamental best practices are recommended:

1. **Develop Software Iteratively:** Plan increments of the system based on customer priorities and develop and deliver the highest priority system features early in the development process.
2. **Manage Requirements**: Explicitly document the customer's requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.
3. **Use Component-Based Architectures:** Structure the system architecture into components.
4. **Visually Model Software:** Use graphical UML models to present static and dynamic views of the software.
5. **Verify Software Quality:** Ensure that the software meets the organisational quality standards.
6. **Control Changes to Software:** Manage changes to the software using a change management system and configuration management procedures and tools.

- The RUP is not a suitable process for all types of development but it does represent a new generation of generic processes. The most important innovations are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Phases are dynamic and have goals. Workflows are static and are technical activities that are not associated with a single phase but may be used throughout the development to achieve the goals of each phase.

EXERCISE

1. What is software engineering? Briefly discuss the need for software engineering.
2. What are attributes of good software? Explain the key challenges facing software engineering
3. What is software engineering? Explain software engineering code of ethics.
4. List and explain ant four software engineering code of ethics.
5. What is ethics doing in a course for software engineers? Explain with the help of case study.
6. How do you solve a case study business ethics? Explain with example.
7. Explain Ethical case study on conflict of interest.
8. With a neat diagram explain the activity model of insulin pump control system.
9. Explain model for Mental Health Care-Patient Management System
10. Design a model to help monitor climate change and to improve the accuracy of weather forecasts
11. Explain the architecture of a digital learning environment.
12. What is a software process model? Explain the types of software process models.
13. With neat block diagram explain generic process model.
14. With the help of diagram explain waterfall model. Also state its advantages and disadvantages.
15. When to use incremental process model?
16. if you have to develop a word processing software product, what process model wills you chose? Justify your answer.
17. What is RAD model- advantages, disadvantages and when to use it?
18. What is the main aim of prototype model?

19. With the help of diagram illustrate spiral model and state two benefits and the problems in spiral development.
20. Explain in detail Concurrent Development Model
21. Give some benefits of using V model?
22. What is the iterative model of the SDLC? What are its advantages and disadvantages?
23. Why Reuse-oriented software engineering is in demand ? Explain in detail.
24. What are four process activities which are common to all software process? Explain it in detail.
25. How can software projects manage change?
26. What is prototyping?
27. What is incremental delivery/development?
28. What is coping with change in software engineering?
29. With a neat diagram explain Boehm's spiral model
30. With neat diagram explain The Rational Unified Process.

UNIT II

AGILE SOFTWARE DEVELOPMENT

2.1 AGILE METHODS

- Agile methodologies are specifically designed to facilitate communication, collaboration, and coordination within a dynamic team environment. This includes practices such as shared team rooms to encourage frequent informal communication, informative workplace environments to allow ubiquitous information dissemination, and rapid feedback cycles through which a software product is evolved incrementally in close partnership with a customer representative. Rather than following to traditionally long periods of upfront requirements gathering and design before software production, agile teams takes feedback early on in the process, and deal with the complexities of software development by practicing rapid iterative development from project inception. In agile methodology, development team focuses on construction rather than design and documentation. The agile methodology uses iterative approach to software development.

In other words agility means:

- Effective (rapid and adaptive) response to change (team members, new technology, requirements).
- Effective communication in structure and attitudes among all team members, technological and business people, software engineers and managers.
- Drawing the customer into the team. Eliminate "us and them" attitude. Planning in an uncertain world has its limits and plan must be flexible.
- Organizing a team so that it is in control of the work performed.
- Eliminate all but the most essential work products and keep them lean.
- Emphasize an incremental delivery strategy as opposed to intermediate products that gets working software to the customer as rapidly as feasible.
- Rapid, incremental delivery of software.
- The development guidelines stress delivery over analysis and design although these activates are not discouraged, and active and continuous communication between developers and customers.

2.1.1 Manifesto for Agile Software Development

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

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

2.1.2 Agility and the Cost of Change

- Conventional wisdom is that the cost of change increases nonlinearly as a project progresses. It is relatively easy to accommodate a change when a team is gathering requirements early in a project. If there are any changes, the costs of doing this work are minimal. But if in the middle of validation testing, a stakeholder is requesting a major functional change then the change requires a modification to the architectural design, construction of new components, changes to other existing components, new testing and so on. Costs rise tremendously.
- A well-designed agile process may "smooth" the cost of change curve by coupling incremental delivery with agile practices such as continuous unit testing and pair programming. Thus team can accommodate changes late in the software project without dramatic cost and time impact.

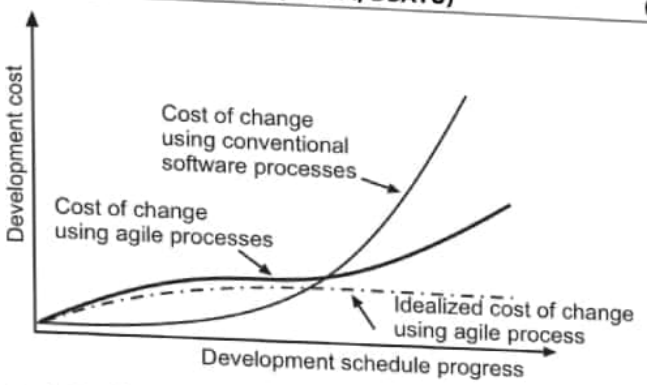


Fig. 2.1 : Change costs as a function of time in development

2.1.3 Agility Principles

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face–to–face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity the art of maximizing the amount of work not done is essential.
- The best architectures, requirements, and designs emerge from self organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

2.2 PLAN-DRIVEN AND AGILE DEVELOPMENT

- Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation. By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity. Fig. 2.2 below shows the distinctions between plan-driven and agile approaches to system specification. In a plan- driven approach, iteration occurs within activities with formal documents (Every project manager should create a small core set of formal documents defining the project objectives, how they are to be achieved, who is going to achieve them, when they are going to be achieved, and how much they are going to cost. These documents may also reveal inconsistencies that are otherwise hard to see) used to communicate between stages of the process. For example, the requirements will evolve and, ultimately, requirements specification will be produced. This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together, rather than separately. A plan-driven software process not necessarily waterfall model plan- driven, incremental development and delivery is possible. It is perfectly feasible to allocate requirements and plan the design and development phase as a series of increments. An agile process is not inevitably code-focused and it may produce some design documentation.

Plan-Driven Development :

- Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
- Iteration occurs within activities.

Agile Development :

- Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process.

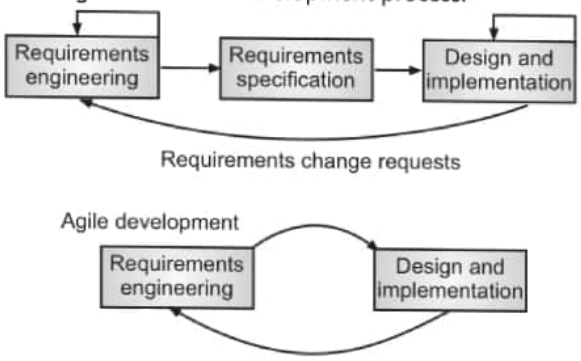


Fig. 2.2: Plan-based and Agile development

- Most projects include elements of plan-driven and agile processes. Deciding the balance depends on Technical, human, organizational issues:
  - ➢ Is it important to have a very detailed specification and design before moving to implementation? If so, software engineer probably needs to use a plan-driven approach.
  - ➢ Is an incremental delivery strategy, where software engineer delivers the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
  - ➢ How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.
  - ➢ What type of system is being developed? Plan-driven approaches may be required for systems that require a lot of analysis be for implementation (e.g. real-time system with complex timing requirements).
  - ➢ What is the expected system lifetime? Long-life time systems may require more design documentation to communicate the original intentions of the system developers to the support team.
  - ➢ What technologies are available to support system development? Agile methods rely on good tools to keep track of an evolving design
  - ➢ How is the development team organized? If the development team is distributed or if part of the development is being outsourced, then software engineer may need to develop design documents to communicate across the development teams.
  - ➢ Are there cultural or organizational issues that may affect the system development? Traditional engineering organizations have a culture of plan-based development, as this is the norm/standard in engineering.
  - ➢ How good are the designers and programmers in the development team? It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code
  - ➢ Is the system subject to external regulation? If a system has to be approved by an external regulator then software engineer will probably be required to produced tailed documentation as part of the system safety case.

Table 2.1 : Key Differences Between Plan-Driven and Aglie Project Management Methodologies

Categories	Plan-Driven	Aglie
Fundamental assumption	Systems are fully specifiable, predictable, and are built through meticulos and extensive planning.	High-quality adaptive software is developed by small teams using the principles of continuous design improvement and testing based on rapid feedback and change.
Management style	Command and control	Leadership and collaboration
Knowledge management	Explicit	Tacit
Communication	Formal	Informal
Development model	Life-cycle model	The evolutionary-delivery model
Desired organizational structure	Mechanistic	Organic
Quality control	Heavy planning and strict control. Late, heavy testing.	Continuous control of requirements, design and solutions. Continuous testing

2.3 EXTREME PROGRAMMING

2.3.1 XP Practices (Principles)

- **Incremental Planning:** Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development.
- **Small Releases:** The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
- **Simple Design:** Enough design is carried out to meet the current requirements and no more.
- **Test-First Development:** An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
- **Refactoring:** All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
- **Pair Programming:** Developers work in pairs, checking each other's work and providing the support to always do a good job.
- **Collective Ownership:** The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
- **Continuous Integration:** As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
- **Sustainable Pace:** Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity.
- **On-Site Customer:** A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

2.3.2 XP Values

There are five XP values communication, simplicity, feedback, courage, and respect.

1. **Communication:** Communication between software engineers and other stakeholders is require to establish required features and functions for the software. XP emphasizes close, yet informal (verbal) collaboration between customers and developers.
2. **Simplicity:** XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code. If the design must be improved, it can be refactored later on.
3. **Feedback:** Feedback is taken from
   (i) The implemented software itself,
   (ii) The customer, and
   (iii) Other software team members.

   As an increment is delivered to a customer, the user stories or use cases that are implemented by the increment are used as a basis for acceptance tests. The function and behavior of the increment is the feedback. Depending upon feedback new requirements are derived as part of iterative planning, the team provides the customer with rapid feedback regarding cost and schedule impact.
4. **Courage and Respect:** Most software teams give way, arguing that "designing for tomorrow" will save time and effort in the long run. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code. By following each of these values, the agile team inculcates respect among members, between stakeholders and team members. As they achieve successful delivery of software increments, the team develops growing respect for the XP process.

2.3.3 XP Process

There are four basic activities of XP process:

1. **Designing**

- It does not mean that XP process exclude designing. Without proper design system becomes complex and projects could scrap. It is important to create a design structure that organizes the logic in the system in such a way that too many dependencies in the system can be avoided.

2. Coding

- In XP coding is considered the only important product of the system development process. XP programmers start to generate codes at the very beginning so "At the end of the day, there has to be a program."

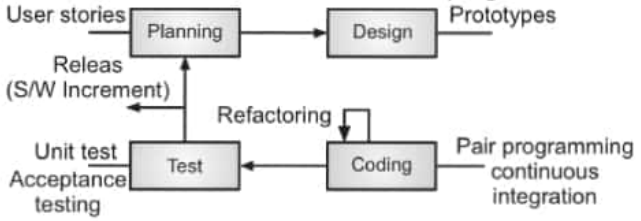


Fig. 2.3 : The Extreme Programming Process

3. Testing

- XP emphasizes to always check if a function works is by testing it. XP uses Unit Tests which are automated tests, and the programmer will write tests as many as possible to try to break the code he or she is writing.

4. Planning/Listening

- Listening ability will enable XP developer to understand what customers want and develop solutions which match customers' needs and desires as close as possible.

In XP these four basic activities are implemented by using following practices:

- **User Stories (Planning):** Customer briefs specification of the new application (features, value, priority) as user story which are viewed as smaller version of use case. These stories will be the base for the project team to do cost estimation and management of the project.
- **Small Releases (Building Blocks):** XP emphasizes on small, simple but frequent versions updates of the application. Each newly added requirement will instantly incorporated and the system is re-released.
- **Metaphor (Standardized Naming Schemes):** Developers and programmers must adhere to standards on names, class names and methods.
- **Collective Ownership:** In XP methodology, all code is considered to be owned by the whole team and not an individual property. Hence, all code is reviewed and updated by everyone.
- **Coding Standard:** Styles and formats of coding must be the same in order to enable compatibility between team members. This approach results in more rapid collaboration.
- **Simple Design:** Always look for system implementation that is as easy as possible implementation of the system yet meets all required functionality.
- **Refactoring:** The application should be continually adjusted and improved by all team members. This requires extremely good communication between members to avoid work duplication.
- **Testing:** Every small release (called building block) must pass tests before being released. In XP, tests are created first and then application code is developed to meet and pass the challenges of those pre-written tests.
- **Pair Programming:** XP programmers work in pairs. All code is developed by two programmers who work together at a single machine. The expectation is that pair programming produces higher quality code at the same or less cost.
- **Continuous Integration:** Software builds are completed several times a day. In this way all developers can avoid work fragmentations because they continuously releasing and integrating code together.
- **40-Hour Workweek:** Keep mental and physical conditions to be up and running by not working more than what the bodies can handle.
- **On-Site Customer:** The customer must be viewed as an integral part of the project. The customer must be arranged to be available at all times in order to ensure that the project is in the right track.

2.3.4 Industrial XP (IXP)

IXP incorporates six new practices that are designed to help and ensure that an XP project works successfully for significant projects within a large organization.

- **Readiness Assessment:** The assessment determines whether
  1. An appropriate development environment exists to support IXP,
  2. The team will be populated by the proper set of stakeholders,
  3. The organization has a distinct quality program and supports continuous improvement,
  4. The organizational culture will support the new values of an agile team, and

5. The broader project community will be populated appropriately.

- **Project Community:** People in the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team. In XP "team" is morph to community. A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders. In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.
- **Project Chartering:** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.
- **Test-Driven Management:** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made up to date. Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.
- **Retrospectives:** An IXP team conducts a specialized technical review after a software increment is delivered called a retrospective. The review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release.
- **Continuous Learning:** Members of the XP team are encouraged to learn new methods and techniques that can lead to a higher quality product.

2.4 AGILE PROJECT MANAGEMENT

- The Agile Project Management (APM) framework is an iterative and cohesive method of effective project management during the entire lifecycle of the project. It successfully integrates the traditional aspects of project management with modern requirements within the scope of the project.
- Rather than focusing only on the development of the final product within the given constraints, the APM framework focuses on delivering both quality and value under the same conditions.
- In the APM Framework, the entire project gets divided into smaller, achievable milestones that are completed in iterations. Quality control is built into the whole process and teams check the quality of their project milestones completed after each cycle.

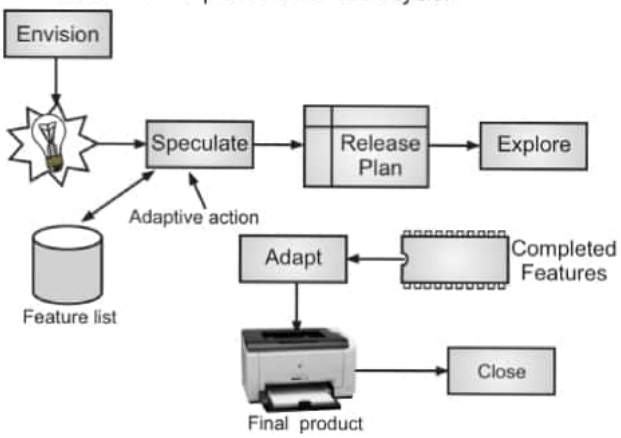


Fig. 2.4 : Agile Project Management Process Framework

1. Envision

- This phase corresponds to the Initiation phase of project management.
- In this phase the team collaborates to brainstorm and define the scope and overall vision of the project. During the Envision segment, key capabilities are outlined, goals and objectives detailed, and participants and stakeholders identified.

2. Speculate

- This phase corresponds to the planning phase of project management. It deals with creating a features list of the final product and how the team would work to achieve it.
- The speculation phase normally revolves around two key activities:
  (i) 'Breaking down' the project into a series of high-level milestones and deciding the expected project timeline.
  (ii) Coming up with the initial understanding of the critical tasks of the project. Priority to a certain class of tasks is given over others in this phase and the team members decide on the way to maintain the quality of the final project deliverable.

3. Explore

- This phase corresponds to the project execution phase of project management. In this phase team members explore the various alternatives to fulfill all the

requirements of the project while staying within the given constraints. The main focus is on creating value and maintaining the quality of the final deliverable.

- Similar to almost all Agile methodologies, teams work by focusing on a single milestone and iterate until perfection is achieved. This phase works parallelly with the Adapt phase because teams may have to change their plan and execution-style if a customer demands it or if the feedback is not as expected.

4. Adapt

- This is perhaps the most distinguishing phase of this framework. The ability to adapt to different circumstances allows the team to be prepared for anything that gets thrown towards them.
- By constantly taking feedback from customers and ensuring that each aspect of the project is up to the end user's requirements, teams can significantly increase their efficiency and effectiveness.

5. Close

- This is the final phase. Teams ensure that the project gets completed in an orderly manner without any hitch. The final deliverable is checked against the updated requirements of the customers and teams ponder over their mistakes in order to avoid them in the future.

2.4.1 Agile Project Management with Scrum

SCRUM is a loose set of guidelines that govern the development process of a product, from its design stages to its completion. It aims to cure some common failures of the typical development process, such as:

- **Chaos Due to Changing Requirements:** The real or perceived requirements of a project usually change drastically from the time the product is designed to when it is released. Under most product development methods, all design is done at the beginning of the project, and then no changes are allowed for or made when the requirements change.
- **Unrealistic Estimates of Time, Cost, and Quality of the Product:** The project management and the developers tend to underestimate how much time and resources a project will take, and how much functionality can be produced within those constraints. In actuality, this usually cannot be accurately predicted at the beginning of the development cycle.
- **Developers are Forced to Lie about how the Project is Progressing:** When management underestimates the time and cost needed to reach a certain level of quality, the developers must either lie about how much progress has been made on the product, or face the indignation of the management.

Scrum is an Agile framework that supports light weight processes that emphasize:

1. Incremental deliveries
2. Quality of product
3. Continuous improvements
4. Discovery of people's potential

1. Scrum Process Flow

- In this process software development life cycle is divided into small parts known as sprints, which have a defined period of time for development. After dividing into parts, the team decides who is going to be responsible for which part of the sprint. Accordingly, in the specified time period, the work is carried out. After the first iteration, Product Owner defines the backlog. The tasks in the backlog are prioritized by the owner (sometimes team might be asked to prioritize the items in the backlog). Then plan is reviewed. The Scrum Master, Product Owner, and the team are members of the sprint planning meeting. The size of each item is determined by the team after the review meeting. This helps in deciding the items which can be completed in the current sprint. This is followed by the work of clearing the backlog. The highest priority items are completed first and the items with lower priority are taken into consideration later. Depending on the policy decided, there might be a daily or a weekly review meeting. This meeting helps in deciding the tasks accomplished as well as the next items to be worked upon. For the items which have not been completed, the problems are discussed and solutions are worked upon. Modifications are done to the sprint backlog either everyday or weekly. This helps create the burn-down chart where the progress made as opposed to the time remaining for the sprint to get over, are charted out. At the end of the sprint, the progress is displayed.
- Finally, the entire team gathers together to review the sprint and find out what went wrong in the preceding

sprint and the change of action that needs to be carried out is also decided. After this starts the next sprint. The advantage of using this approach for computer software development is the increase in productivity. The continuous reviews help in overall improvement. Better communication is observed between the customer and the software developing team. The end result of all this is a superior software product.

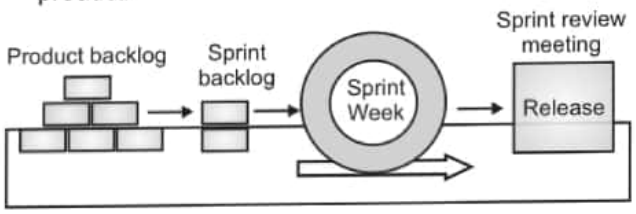


Fig. 2.5 : Scrum Process Flow

2. Scrum Roles

(i) Product Owner :

- Sets the direction of the product
- Represents the stakeholders and business
- Make sure right features making into the product backlog
- Basically responsible for product backlog and prioritization

(ii) Scrum Master :

- Make sure project is running smoothly
- Sets up the meetings
- Monitoring and assessing the project progress
- Facilitates the release planning
- Responsible to protect the team from other interferences and clears the way

(iii) Scrum Team

- A cross-functional group of 8-10 people who do the actual analysis, design, implementation, testing etc.,
- The team members estimate the task and commit to delivery
- Individuals empowered to take decisions with help of team members
- Scrum Team consists of Programmers, Interface Designers, Testers, Technical Writers, Technical Architect and Product Owner

3. Scrum Life Cycle

Following steps are followed in scrum based agile software development:

1. First the high level requirements are finalized based on end user perspective/opinions; hence it is called user stories. These user stories are broken up into tasks till we will not be able to break them down further. This is created in a document called as product backlog by Product Owner in association with customer. All the stories/ tasks are prioritized.

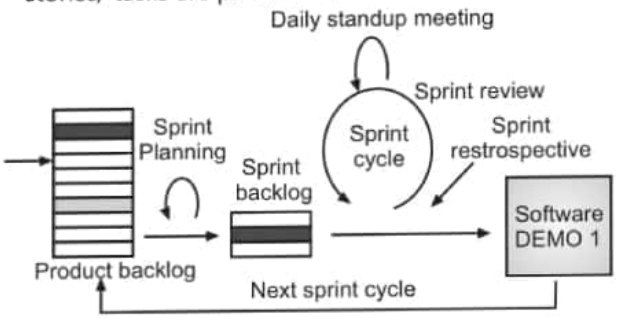


Fig. 2.6 : Scrum Life Cycle

2. These tasks after prioritization are divided into sprints. Each sprint will contain set of tasks. This is done in sprint planning. The document created is called sprint backlog.
3. Now the Sprint process will come in execution phase where in the scrum team will do mainly development/testing activities. During sprint execution phase scrum team daily meets to discuss the progress. These meetings are facilitated by scrum master. In these meetings each team members will talk on three questions i.e. what they have achieved till yesterday, what they are going to do and are there any impediments on the way of smooth work progress.
4. After the end of sprint scrum team will do sprint review and also sprint retrospectives. This helps in better planning for next sprints.
5. After the sprint retrospectives the phase 1 or iteration 1 of the development carried out. It is evaluated with definition of DONE which is also called as acceptance criteria. Now the increment 1 of the development carried out will be demonstrated to the customer and feedback will be obtained from the customer. The same feedback will be utilized for next sprints.

- The steps 1 to 5 mentioned above are revisited again in the consecutive sprints. Hence the scrum life cycle is a continuous process.

4. Scrum Product Backlog

- The agile product backlog in Scrum is a prioritized features list, containing short descriptions of all functionality desired in the product. When applying Scrum, it's not necessary to start a project with a lengthy, upfront effort to document all requirements. Scrum team and its product owner begin by writing down everything they can think of for agile backlog prioritization. This agile product backlog is enough for starting first sprint. The Scrum product backlog is then allowed to grow and change as more is learned about the product and its customers.
- A typical Scrum backlog comprises the following different types of items:
  - ➢ Features
  - ➢ Bugs
  - ➢ Technical work
  - ➢ Knowledge acquisition

5. Sprint Planning Meeting

- In Scrum, the sprint planning meeting is attended by the product owner, Scrum Master and the entire Scrum team.
- During the sprint planning meeting, the product owner describes the highest priority features to the team. The team asks enough questions that they can turn a high-level user story of the product backlog into the more detailed tasks of the sprint backlog.
- The product owner doesn't have to describe every item being tracked on the product backlog. A good guideline is for the product owner to come to the sprint planning meeting prepared to talk about two sprint's worth of product backlog items. For example: suppose a team always finishes five product backlog items. Their product owner should enter the meeting prepared to talk about the top 10 priorities.
- There are two defined artifacts that result from a sprint planning meeting:
  - (i) A sprint goal
  - (ii) A sprint backlog
- A sprint goal is a short (one or two sentence) description of what the team plans to achieve during the sprint. It is written collaboratively by the team and the product owner. The following are example sprint goals on an e-Commerce application:
  - ➢ Implement basic shopping cart functionality including add, remove, and update quantities.
  - ➢ Develop the checkout process: pay for an order, pick shipping, order gift wrapping, etc.
- The sprint goal can be used for quick reporting to those outside the sprint.
- The sprint backlog is the other output of sprint planning. A sprint backlog is a list of the product backlog items the team commits to delivering plus the list of tasks necessary to delivering those product backlog items. Each task on the sprint backlog is also usually estimated.
- An important point to be discussed is that team selects how much work they can do in the next sprint.

6. Sprint Backlog

- The Sprint Backlog contains user stories popped off the top of the Product Backlog during Sprint Planning. Sprint Backlog generally contains many things which are not on the Product Backlog, such as:
  - ➢ Tasks that have been decomposed from the user stories accepted by the Team for the current iteration.
  - ➢ Story points or time estimates for individual tasks.
  - ➢ Refinements of the "definition of done" as it relates to a specific story or task.
  - ➢ Refinements to stories that don't compromise the Sprint Goal or require the Product Owner to call for an early termination of the Sprint.
  - ➢ In-sprint stories or tasks added by the Team to support the current Sprint Goal.
- The Sprint Goal and stories accepted for the current Sprint are set during Spring Planning, but tasks on the Sprint Backlog are updated and modified.

7. Sprint Execution

- Sprint execution is the work the Scrum team performs to meet the sprint goal. It begins after sprint planning and ends when the sprint review starts. During sprint planning the team produces a plan for how to achieve the sprint goal. Most teams create a sprint backlog, which typically lists product backlog items and their associated tasks and estimated effort-hours. Execution focuses on: (here we mean team).

- ➢ All tasks to get to production ready completed? Did we get done everything that made up our definition of done?
- ➢ Defects affecting acceptance criteria are resolved? If a defect is found and impacts the acceptance criteria, do we resolve it?
- ➢ Work focused on highest priority stories? Does the team focus on completing the highest priority stories first or do we start every story all at once?
- ➢ Dependencies, risks and issues identified and acted upon? If we find a dependency or issue in the sprint, do we consider its impact and take action to ensure completion of the highest priority stories first?

- Focusing on these, however, will greatly increase the team's performance and ability to improve time to market, reduce defects and increase predictability while implementing agile development.

8. Daily Scrum Meeting

Each day during the sprint, a project team communication meeting occurs. This is called a Daily Scrum (meeting) and has specific guidelines:

- All members of the development team come prepared with the updates for the meeting.
- The meeting starts precisely on time even if some development team members are missing.
- The meeting should happen at the same location and same time every day.
- The meeting length is set (time-boxed) to 15 minutes.
- All are welcome, but normally only the core roles speak.

During the meeting, each team member answers three questions:

(i) What have you done since yesterday?

(ii) What are you planning to do today?

(iii) Any obstacle/uncertain blocks?

Any obstacle/uncertain block identified in this meeting is documented by the Scrum Master and worked towards resolution outside of this meeting. No detailed discussions shall happen in this meeting.

9. Maintaining Sprint Backlog and Burndown Chart

- The sprint backlog is a list of tasks identified by the Scrum team to be completed during the Scrum sprint. During the sprint planning meeting, the team selects some number of product backlog items, usually in the form of user stories, and identifies the tasks necessary to complete each user story. Most teams also estimate how many hours each task will take someone on the team to complete.
- It's critical that the team selects the items and size of the sprint backlog. Because they are the people committing to completing the tasks, they must be the people to choose what they are committing to during the Scrum sprint.
- The sprint backlog is commonly maintained as a spreadsheet, but it is also possible to use defect tracking system or any of a number of software products designed specifically for Scrum or agile. An example of a sprint backlog in a spreadsheet looks like this:

User Story	Tasks	Day 1	Day 2	Day 3	Day 4	.....
Users high priority requirement	Design....	6	5	6	4	
	Code....	6	8	6	5	
	Test....	4	3	4	4	
	Meet Customer....	4	2	4	4	
Users modification	Design solution...	6	6	6	6	
	Update code...	8	6	8	8	
	Write test plan...	4	4	4	4	
	Get feedback...	4	4	4	4	

- During the Scrum sprint, team members are expected to update the sprint backlog as new information is available, but minimally once per day. Many teams will do this during the daily scrum. Once each day, the estimated work remaining in the sprint is calculated and graphed by the Scrum Master, resulting in a sprint burn-down chart like this one:

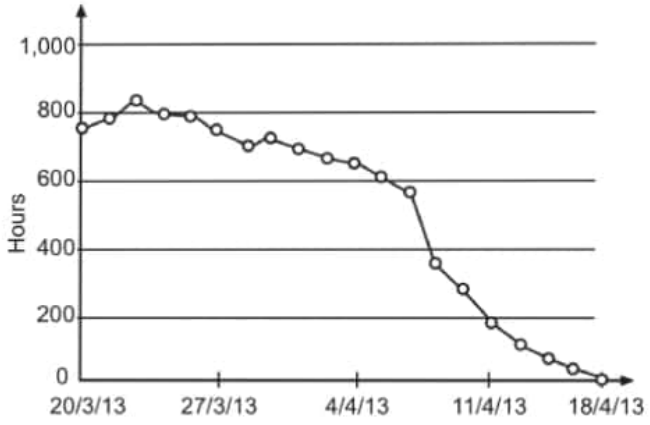


Fig. 2.7 : Sprint Burndown Chart

- The sprint burndown chart is a publicly displayed chart showing remaining work in the sprint backlog. Updated every day, it gives you simple view of the sprint progress.
- Sprint burndown chart is one of the agile tools used by scrum teams to monitor and assess progress in a sprint, adding additional scope in the middle of the sprint shows in the sprint burndown chart. Similarly, de-scoping of the work is also reflected in the sprint burndown chart. Velocity of the team can be related to the gradient of the burndown chart or numbers of hours of work completed daily by the team
- Let's take an example using the sprint burndown chart above. As you can see, the team in this scenario pulled in too much work initially into the sprint backlog, and still had nearly 600 hours to go on 27/3/13. Let's say in this case, the product owner was consulted and agreed to remove some work from the sprint. This resulted in the big drop on the chart between 27/3/13 (619 hours) and 28/3/13. From there, the team made consistent progress and finished the Scrum sprint successfully.

10. Sprint Review and Retrospective

- **Sprint Review** is for everyone involved, especially stakeholders, to inspect where the project is and discuss how to adapt as needed. Sprint review revolves around what was built the "shippable product increment" produced in the last sprint and the overall product, not how it was produced. It is good if Product Owner "represents" stakeholders.
- **Sprint Retrospective** is primarily for the team to inspect their last sprint, concentrating less on what was done than on how it was done, and then adapt their way of work. It is better not to include management for retrospective. But having said that an overall retrospective on the project or on a longer chunk of it (like a quarter or half a year) that would include management may make sense, but it would not necessarily include all team members (if you have many teams that would make it even impossible to do). Such a retrospective would concentrate on "big picture" and could be very beneficial – if there is of course a right atmosphere within company for people to be honest enough in such a retrospective for results to be useful.

2.5 SCALING AGILE METHODS

- Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team. It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together. Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

Two perspectives on scaling of agile methods:

1. 'Scaling Up'

- Using agile methods for developing large software systems that cannot be developed by a small team. For large systems development, it is not possible to focus only on the code of the system; you need to do more up-front design and system documentation. Cross-team communication mechanisms have to be designed and used, which should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible; however, it is essential to maintain frequent system builds and regular releases of the system.

2. 'Scaling Out'

- How agile methods can be introduced across a large organization with many years of software development experience. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach. Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Practical Problems with Agile Methods

- In some areas, particularly in the development of software products and apps, agile development has been incredibly successful. It is by far the best approach to use for this type of system. However, agile methods may not be suitable for other types of software development, such as embedded systems engineering or the development of large and complex systems.
- For large, long-lifetime systems that are developed by a software company for an external client, using an agile approach presents a number of problems.
  - ➢ The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
  - ➢ Agile methods are most appropriate for new software development rather than for software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
  - ➢ Agile methods are designed for small co-located teams, yet much software development now involves worldwide distributed teams.
- Moreover, where maintenance involves a custom system that must be changed in response to new business requirements, there is no clear consensus on the suitability of agile methods for software maintenance Three types of problems can arise:
  1. Lack of product documentation
  2. Keeping customers involved
  3. Development team continuity

Table 2.2

Principle	Practice
Customer involvement	This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development. Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.
Embrace change	Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
Incremental delivery	Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know product features several months in advance to prepare an effective marketing campaign.
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People, not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods and therefore may not interact well with other team members.

Agile and Plan-Driven Methods

- Agile fundamentals: Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.
- The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
- Agile methods are most appropriate for new software development rather than software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
- Agile methods are designed for small co-located teams yet much software development now involves worldwide distributed teams.
- Most software contracts for custom systems are based around a specification, which sets out what has to be implemented by the system developer for the system customer.
- It is not therefore possible to interleave specification and development as recommended in agile development.
- A contract that pays for developer time rather than functionality is seen as a high risk by many legal departments because what has to be delivered cannot be guaranteed.

- 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?
- If agile methods are to be successful, they have to support maintenance as well as original development. Agile development relies on the development team knowing and understanding what has to be done. For long-lifetime systems, this is a real problem as the original developers will not always work on the system. Scaling agile requires a mix of agile and plan-based development.
- Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
- Are customer representatives available and willing to work closely with the development team?
- How large is the system that is being developed?
- Agile methods minimize documentation but documentation may be essential for distributed teams.

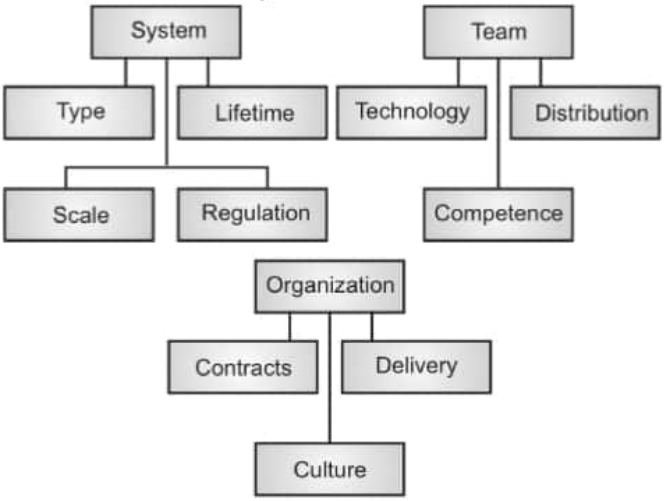


Fig. 2.8 : Agile and plan-based factors

- Agile methods are most effective a relatively small co-located team who can communicate informally.
- Systems that require a lot of analysis before implementation need a fairly detailed design to carry out this analysis.
- Long-lifetime systems require documentation to communicate the intentions of the system developers to the support team.
- If a system is regulated you will probably be required to produce detailed documentation as part of the system safety case.
- It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code.
- Design documents may be required if the team is distributed.
- IDE support for collaborative work is essential for distributed teams.
- Can the organization adapt to different kinds of development contract or does the contracts department insist on standardization?
- Does the culture support individual initiative which is an inherent part of agile development?
- Do customers want incremental delivery and involvement in the process?

Agile Methods for Large Systems

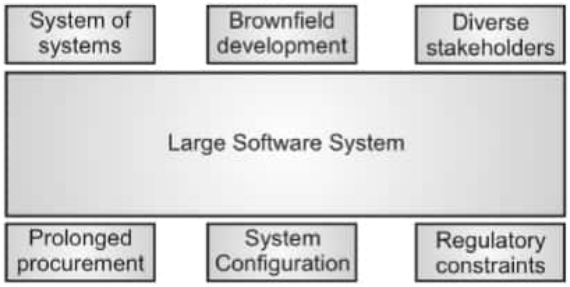


Fig. 2.9: Factors in large systems

- Large systems are usually collections of separate, communicating systems, where separate teams develop each system. These teams may be working in different places, sometimes in different time zones.
- Changing systems that are part of an SoS may be impossible as this may affect their principal purpose. There can't be a single stakeholder representative and many stakeholders may have to be involved in requirements refinement.
- Large systems are 'brown field systems', that is they include and interact with a number of existing systems.
- Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development.
- Large systems usually have a diverse set of stakeholders. It is practically impossible to involve all of

these different stakeholders in the development process.

- Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- Regulators may be able to stop a non-compliant system being deployed and used.
- Regulation and compliance issues apply to non-critical as well as safety-critical systems.
- Following defined processes or using particular methods may be required by the regulator where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. Partial configuration may be impossible as it may not implement the required user functionality.
- Incremental development by configuration may therefore be impossible. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects. Approaches to scaling agile methods for large systems all focus on integrating agile methods into plan- based approaches

IBM's Agility at Scale Model

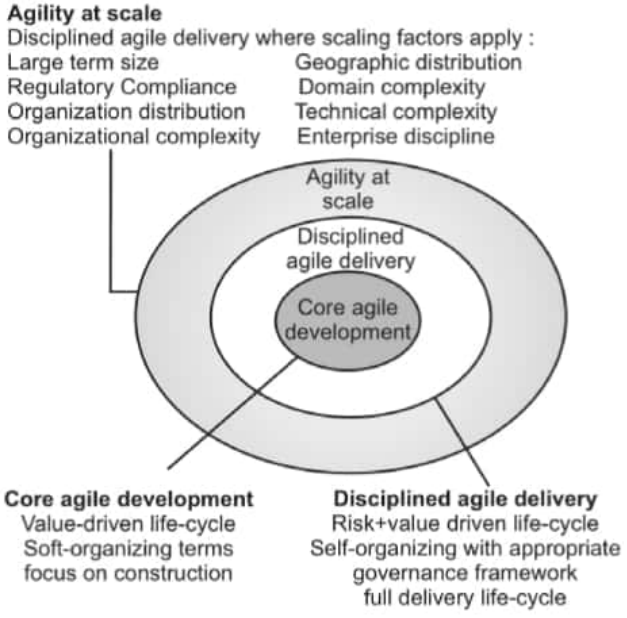


Fig. 2.10

Core Agile Development

- Maintaining agile principles where focus is on customer value, implementation rather than documentation and team responsibility

Disciplined Agile Delivery

- Elements of plan-based development introduced. More focus on risk and recognition of documentation requirements

Agility at Scale

- Recognition of importance of complexity and complexity management
  - ➢ A completely incremental approach to requirements engineering is impossible.
  - ➢ There needs to be a complete overall picture of the system requirements that may then be refined by stakeholder representatives
  - ➢ There cannot be a single product owner or customer representative.
  - ➢ For large systems development, it is not possible to focus only on the code of the system.
  - ➢ Cross-team communication mechanisms have to be designed and used.
  - ➢ Continuous integration is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.
- Using agile methods for large systems engineering means integrating agile practices with the engineering practices used in large systems development

2.6 REQUIREMENTS ENGINEERING

- The requirements for a system are the descriptions of the services that a system should provide and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called Requirements Engineering (RE).
- Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description.
- I distinguish between them by using the term user requirements to mean the high-level abstract

requirements and system requirements to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

- User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.
- System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Functional and Non-Functional Requirements

- Software system requirements are often classified as functional or non-functional requirements:

Functional Requirements:

- These are the requirements that the end user specifically demands as basic facilities that the system should offer. All these functionalities need to be necessarily incorporated into the system as a part of the contract. These are represented or stated in the form of input to be given to the system, the operation performed and the output expected. They are basically the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.

Non-Functional Requirements:

- These are the constraints or the requirements imposed on the system. They specify the quality attribute of the software. Non-Functional Requirements deal with issues like scalability, maintainability, performance, portability, security, reliability, and many more. Non-Functional Requirements address vital issues of quality for software systems. If NFRs not addressed properly, the results can include:
  - Users, clients, and developers are unsatisfied.
  - Inconsistent software.
  - Time and cost overrun to fix the software which was prepared without keeping NFRs in mind.

Types of Non-Functional Requirement :

1. Scalability
2. Reliability
3. Regulatory
4. Maintainability
5. Serviceability
6. Utility
7. Security
8. Manageability
9. Data integrity
10. Capacity
11. Regulatory
12. Availability
13. Usability
14. Interoperability
15. Environmental

These can be Classified as :

1. **Performance Constraints :** Reliability, security, response time, etc.
2. **Operating Constraints :** These include physical constraints (size, weight), personnel availability, skill level considerations, system accessibility for maintenance, etc.
3. **Interface Constraints :** These describe how the system is to interface with its environment, users, and other systems. For example, user interfaces and their qualities (e.g., user-friendliness).
4. **Economic Constraints :** Immediate and/or long-term costs.
5. **Lifecycle Requirements :** Quality of the design: These measured in terms such as maintainability, enhance ability, portability.

Advantages of Non-Functional Requirement :

- They ensure the software system follows legal and adherence rules.
- They specify the quality attribute of the software.
- They ensure the reliability, availability, performance, and scalability of the software system
- They help in constructing the security policy of the software system.

- They ensure good user experience, ease of operating the software, and minimize the cost factor.

Disadvantages of Non-Functional Requirement :

- The nonfunctional requirement may affect the various high-level software subsystem.
- They generally increase the cost as they require special consideration during the software architecture/high-level design phase.
- It is difficult to change or alter non-functional requirements once you pass them to the architecture phase.

Table 2.3

Functional Requirements	Non Functional Requirements
A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
It specifies "What should the software system do?"	It places constraints on "How should the software system fulfill the functional requirements?"
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.
It is captured in use case.	It is captured as a quality attribute.
Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
Usually easy to define.	Usually more difficult to define.
Example	**Example**
1. Authentication of user whenever he/she logs into the system.	1. Emails should be sent with a latency of no greater than 12 hours from such an activity.
2. System shutdown in case of a cyber attack.	2. The processing of each request should be done within 10 seconds
3. A Verification email is sent to user whenever he/she registers for the first time on some software system.	3. The site should load in 3 seconds when the number of simultaneous users are > 10000

2.7 THE SOFTWARE REQUIREMENTS DOCUMENT

- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Agile Methods and Requirements

- Many agile methods argue that producing a requirements document is a waste of time as requirements change so quickly.
- The document is therefore always out of date.
- Methods such as XP use incremental requirements engineering and express requirements as 'user stories' This is practical for business systems but problematic for systems that require a lot of pre-delivery analysis (e.g. critical systems) or systems developed by several teams.

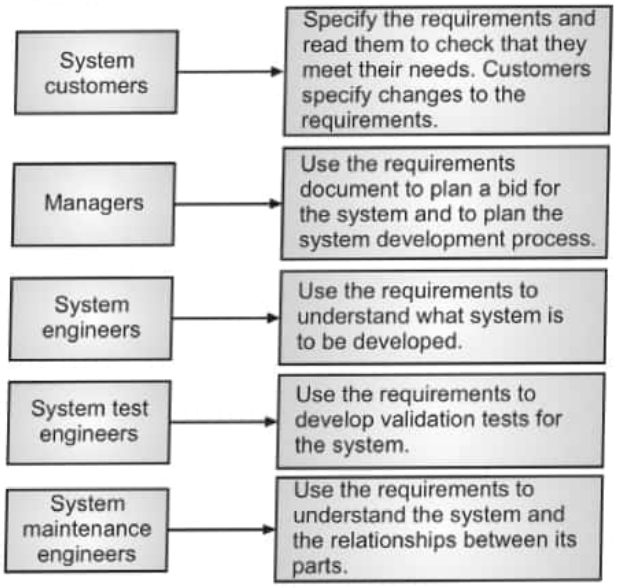


Fig. 2.11: Users of Requirements Document

- The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Fig. 2.11 shows possible users of the document and how they use it.

Requirements Document Variability:

- Information in requirements document depends on type of system and the approach to development used.
- Systems developed incrementally will, typically, have less detail in the requirements document.
- Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

Table 2.4 : The Structure of a Requirements Document

Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

- Software Requirement Specification SRS is a document created by the system analyst after the requirements are collected from various stakeholders.
- SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.
- The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should Come up with Following Features:

- User requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

2.8 REQUIREMENTS SPECIFICATION

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

Requirements and Design

- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are in separable
  - ➢ A system architecture may be designed to structure the requirements;
  - ➢ The system may inter-operate with other systems that generate design requirements;
  - ➢ The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
  - ➢ This may be the consequence of a regulatory requirement.

Table 2.5: Ways of Writing a System Requirements Specification

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

1. Natural Language Specification

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

Guidelines for Writing Requirements

To minimize misunderstandings when writing natural language requirements, follow these simple guidelines:

- Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. I suggest that, wherever possible, you should write the requirement in one or two sentences of natural language.
- Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using "shall." Desirable requirements are not essential and are written using "should."
- Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
- Do not assume that readers understand technical, software engineering language. It is easy for words such as "architecture" and "module" to be

misunderstood. Wherever possible, you should avoid the use of jargon, abbreviations, and acronyms.

- Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and who proposed the requirement (the requirement source), so that you know whom to consult if the requirement has to be changed. Requirements rationale is particularly useful when requirements are changed, as it may help decide what changes would be undesirable

2. Structured Specifications

- An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.
- This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.
- When a standard format is used for specifying functional requirements, the following information should be included:
  - ➢ A description of the function or entity being specified.
  - ➢ A description of its inputs and the origin of these inputs.
  - ➢ A description of its outputs and the destination of these outputs.
  - ➢ Information about the information needed for the computation or other entities in the system that are required (the "requires" part).
  - ➢ A description of the action to be taken.
  - ➢ If a functional approach is used, a precondition setting out what must be true before the function is called, and a post condition specifying what is true after the function is called.
  - ➢ A description of the side effects (if any) of the operation.

Example 1 : Form-Based Specifications

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

Example 2 :Tabular Specification

- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.
- For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

2.9 REQUIREMENTS ENGINEERING PROCESSES

- Requirement Engineering is the process of defining, documenting and maintaining the requirements. It is a process of gathering and defining service provided by the system. Requirements Engineering Process consists of the following main activities:
  1. Requirements elicitation
  2. Requirements specification
  3. Requirements verification and validation
  4. Requirements management

1. Requirements Elicitation:

- It is related to the various ways used to gain knowledge about the project domain and requirements. The various sources of domain knowledge include customers, business manuals, the existing software of same type, standards and other stakeholders of the project.

  The techniques used for requirements elicitation include interviews, brainstorming, task analysis, Delphi technique, prototyping, etc. Elicitation does not produce formal models of the requirements understood. Instead, it widens the domain knowledge of the analyst and thus helps in providing input to the next stage.

2. Requirements Specification:

- This activity is used to produce formal software requirement models. All the requirements including the functional as well as the non-functional requirements and the constraints are specified by these models in totality. During specification, more knowledge about the problem may be required which

can again trigger the elicitation process. The models used at this stage include ER diagrams, Data Flow Diagrams (DFDs), Function Decomposition Diagrams (FDDs), data dictionaries, etc.

3. Requirements Verification and Validation:

- **Verification:** It refers to the set of tasks that ensures that the software correctly implements a specific function.
- **Validation:** It refers to a different set of tasks that ensures that the software that has been built is traceable to customer requirements. If requirements are not validated, errors in the requirement definitions would propagate to the successive stages resulting in a lot of modification and rework.

The main steps for this process include:

- ➢ The requirements should be consistent with all the other requirements i.e. no two requirements should conflict with each other.
- ➢ The requirements should be complete in every sense.
- ➢ The requirements should be practically achievable.

Reviews, buddy checks, making test cases, etc. are some of the methods used for this.

4. Requirements Management:

- Requirement management is the process of analyzing, documenting, tracking, prioritizing and agreeing on the requirement and controlling the communication to relevant stakeholders. This stage takes care of the changing nature of requirements. It should be ensured that the SRS is as modifiable as possible so as to incorporate changes in requirements specified by the end users at later stages too. Being able to modify the software as per requirements in a systematic and controlled manner is an extremely important part of the requirements engineering process.
- Requirements engineering is an iterative process in which the activities are inter leaved. Fig. 2.12 shows this interleaving. The activities are organized as an iterative process around a spiral. The output of the RE process is a system requirements document. The amount of time and effort devoted to each activity in an iteration depends on the stage of the overall process, the type of system being developed, and the budget that is available.

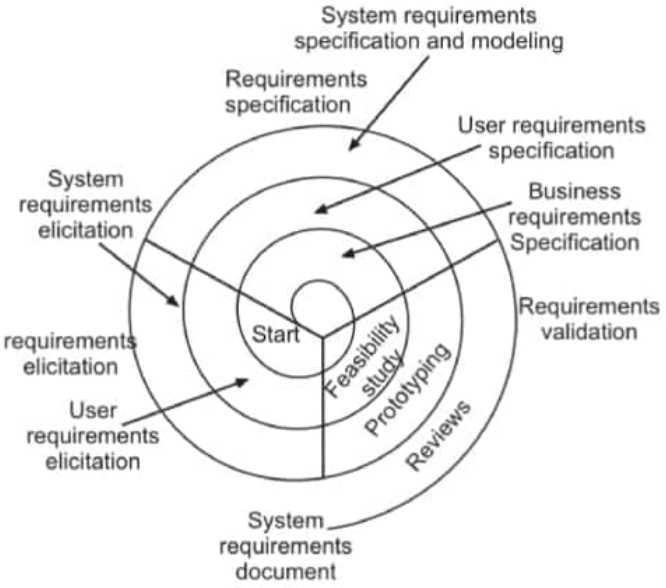


Fig. 2.12 : A spiral view of the requirements engineering process

- Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the non-functional requirements and more detailed system requirements.

2.10 REQUIREMENTS ELICITATION AND ANALYSIS

- The aims of the requirements elicitation process are to understand the work that stakeholders do and how they might use a new system to help support that work.
- During requirements elicitation, software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, and so on. Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:
  - ➢ Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.

- ➢ Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
- ➢ Different stakeholders, with diverse requirements, may express their requirements in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
- ➢ Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
- ➢ The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from news take holders who were not originally consulted.

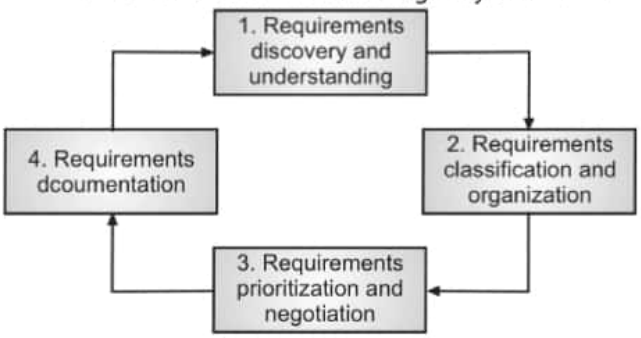


Fig. 2.13: Requirements elicitation and analysis process

- Each organization will have its own version or instantiation of this general model, depending on local factors such as the expertise of the staff, the type of system being developed, and the standards used.

The process activities are:

1. Requirements Discovery and Understanding

- This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.

2. Requirements Classification and Organization

- This activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.

3. Requirements Prioritization and Negotiation

- Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. Requirements Documentation

- The requirements are documented and input into the next round of the spiral. An early draft of the software requirements documents may be produced at this stage, or the requirements may simply be maintained in formally on whiteboards, wikis, or other shared spaces.
- Fig. 2.13 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation.
- The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document has been produced.

2.10.1 Requirements Elicitation Techniques

1. Interviews

- Interviews are probably the most traditional and commonly used technique for requirements elicitation.
- Human based social activities, they are inherently informal and their effectiveness depends greatly on the quality of interaction between the participants.
- Provide an efficient way to collect large amounts of data quickly.
- The results of interviews, such as the usefulness of the information gathered, can vary significantly depending on the skill of the interviewer
- Three types of interviews :

  (i) Unstructured interviews.

  (ii) Structured interviews.

  (iii) Semi structured interviews.

  The latter generally representing a combination of the former two.

Unstructured Interviews

- Do not follow a predetermined agenda or list of questions
- There is the risk that some topics may be completely neglected
- Focus in too much detail on some areas
- Best applied for exploration when there is a limited understanding of the domain, or as a precursor to more focused and detailed structured interviews.

Structured Interviews

- Conducted using a predetermined set of questions. The success of structured interviews depends on
  - ➢ Knowing what are the right questions to ask,
  - ➢ When should they be asked, and
  - ➢ Who should answer them.
- Templates that provide guidance on structured interviews for requirements elicitation such as used
- Although structured interviews tend to limit the investigation of new ideas, they are generally considered to be rigorous and effective.

2. Ethnography

- Ethnography being the study of people in their natural setting, involves the analyst actively or passively participating in the normal activities of the users over an extended period of time whilst collecting information on the operations being performed.
- These techniques are especially useful when addressing contextual factors such as usability, and when investigating collaborative work settings where the understanding of interactions between different users with the system is paramount.
- In practice, ethnography is particularly effective when the need for a new system is a result of existing problems with processes and procedures, and in identifying social patterns and complex relationships between human stakeholders.
- Ethnography is particularly effective for discovering two types of requirements:
  - (i) Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work. In practice, people never follow formal processes. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. The conflict alert system is sensitive and issues audible warnings even when planes are far apart. Controllers may find these distracting and prefer to use other strategies to ensure that planes are not on conflicting flight paths.
  - (ii) Requirements derived from cooperation and awareness of other people's activities. For example, Air Traffic Controllers (ATCs) may use an awareness of other controllers' work to predict the number of aircraft that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.
- Ethnography can be combined with the development of a system prototype. The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study.

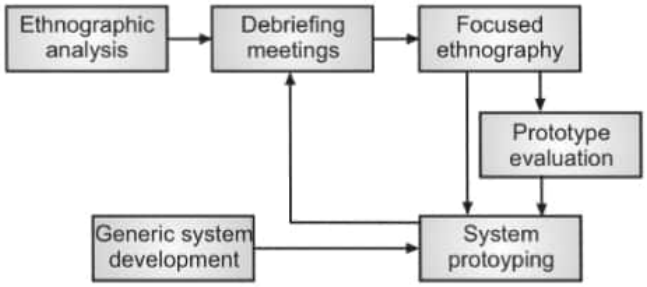


Fig. 2.14 : Ethnography and prototyping for requirements analysis

- Ethnography is helpful to understand existing systems, but this understanding does not always help with innovation. Innovation is particularly relevant for new product development. Commentators have suggested that Nokia used ethnography to discover how people used their phones and developed new phone models on that basis; Apple, on the other hand, ignored current use and revolutionized the mobile phone industry with the introduction of the iPhone.

3. Scenarios

- Scenarios are narrative and specific descriptions of current and future processes including actions and interactions between the users and the system.
- Like use cases, scenarios do not typically consider the internal structure of the system, and require an incremental and interactive approach to their development.
- structured and rigorous approaches to requirements elicitation using scenarios including: CREWS, The Inquiry Cycle, SBRE and Scenario Plus
- Very useful for understanding and validating requirements, as well as test case development.

2.11 REQUIREMENTS VALIDATION

- Requirements validation is the process of checking that requirements define the system that the customer really wants. It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements. Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.
- The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. A change to the requirements usually means that the system design and implementation must also be changed.
- Furthermore, the system must then be retested.
- During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document.
- These checks include:

1. Validity Checks

- These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.

2. Consistency Checks

- Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. Completeness Checks

- The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. Realism Checks

- By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. These checks should also take account of the budget and schedule for the system development.

5. Verifiability

- To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.
- This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.
- A number of requirements validation techniques can be used individually or in conjunction with one another:

  ➢ **Requirements Reviews :** The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

  ➢ **Prototyping :** This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.

  ➢ **Test-Case Generation**: Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of test-driven development.

2.12 REQUIREMENTS MANAGEMENT

- Requirements management is a systematic approach to eliciting, organizing, and documenting the requirement of the system, and a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system. Requirements management includes all activities intended to maintain the integrity and accuracy of expected requirements.
  - ➢ Manage changes to agreed requirements
  - ➢ Manage changes to baseline (increments)
  - ➢ Keep project plans synchronized with requirements
  - ➢ Control versions of individual requirements and versions of requirements documents
  - ➢ Manage relationships between requirements
  - ➢ Managing the dependencies between the requirements document and other documents produced in the systems engineering process
  - ➢ Track requirements status

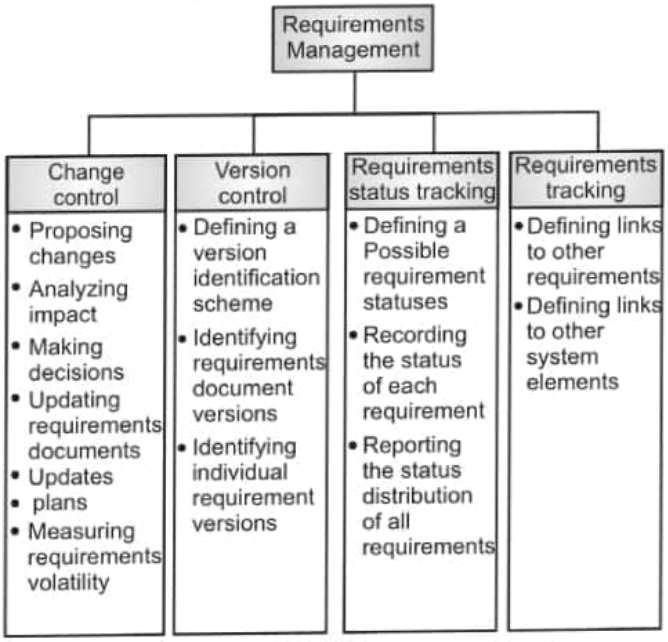


Fig. 2.15 : Requirements management Activities

Requirements Management Planning

- Requirements management planning is concerned with establishing how a set of evolving requirements will be managed. During the planning stage, we have to decide on a number of issues:
  - ➢ **Requirements Identification :** Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
  - ➢ **A Change Management Process :** This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
  - ➢ **Traceability Policies :** These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
  - ➢ **Tool Support :** Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to shared spreadsheets and simple database systems.
- Requirements management needs automated support, and the software tools for this should be chosen during the planning phase. We need tool support for:
  - ➢ **Requirements Storage :** The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
  - ➢ **Change Management** : The process of change management is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.
  - ➢ **Traceability Management** : As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

Requirements Change Management

- Requirements change management should be applied to all proposed changes to a system's requirements after the requirements document has been approved.

- Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

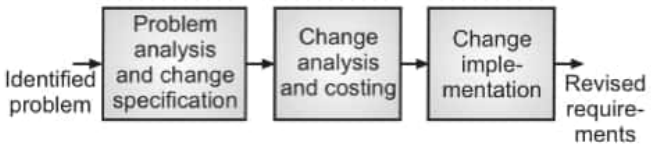


Fig. 2.16 : Requirements change management

There are three principal stages to a change management process:

1. **Problem Analysis and Change Specification :** The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
2. **Change Analysis and Costing :** The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made as to whether or not to proceed with the requirements change.
3. **Change Implementation :** The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

EXERCISE

1. State Agile manifesto.
2. Explain agility and cost of change.
3. What are the Principles of Agile developments?
4. Explain plan driven development with a neat block diagram.
5. Explain plan-based development and agile development.
6. State XP practice principles.
7. What are XP values?
8. How XP process helps in project development?
9. Write short note on
   1. pair programming
   2. Refactoring
   3. Testing in XP
10. What are industrial XP practices? Explain.
11. What is Agile Project Management?
12. Explain Scrum Process Flow with neat diagram.
13. Explain Scrum life cycle with neat diagram.
14. What is Scrum Sprint?
15. Explain product backlog in Scrum
16. What is the role of scrum master in Scrum?
17. What is scrum burn down chart ?
18. Explain the activities performed in daily scrum meeting
19. Explain : Sprint Review and Retrospective
20. How the agile methods are scaled? State the coping of agile methods for large system engineering.
21. What is requirement engineering. Distinguish functional and non functional requirements.
22. Explain the structure of requirement document.
23. Explain different ways of writing a system requirements specification.
24. With a neat diagram explain the different stages of requirement engineering process. What are its benefits?

25. Explain Requirement elicitation and analysis process.
26. Write short note on interviewing.
27. Explain Ethnography in detail.
28. How do you validate requirements?
29. What is requirements management? Why it is important?
30. What are the steps in requirements change management process?

✠ ✠ ✠

UNIT III

SYSTEM MODELING

3.1 INTRODUCTION

- System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
  - ➢ System modeling representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
  - ➢ System modeling helps the analyst to understand the functionality of the system and models are used to communicate with end users.
- Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
  - ➢ Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
  - ➢ In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

3.2 CONTEXT MODELS

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries. Social and organizational concerns may affect the decision on where to position system boundaries. Context models simply show the other systems in the environment, not how the system being developed is used in that environment
- The Unified **Modeling** Language as used in systems engineering defines a **context model** as the physical scope of the system being designed, which could include the user as well as the environment and other actors.
- At specification of a system, user should decide on the system boundaries, that is, on what is and is not part of the system being developed. This involves working with system stakeholders to decide what functionality should be included in the system and what processing and operations should be carried out in the system's operational environment.
- You should look in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.
- For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.
- For example, specification for the ATM information system to manage information about all activities of money withdrawing process.
- In the specification for this system, you have to decide whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about customer) or whether it should also collect individual information. The advantage of relying on other systems for ATM Information is that you avoid duplicating data.

- The major disadvantage, however, is that using other systems may make it slower to access information, and if these systems are unavailable, then it may be impossible to use the ATM system.

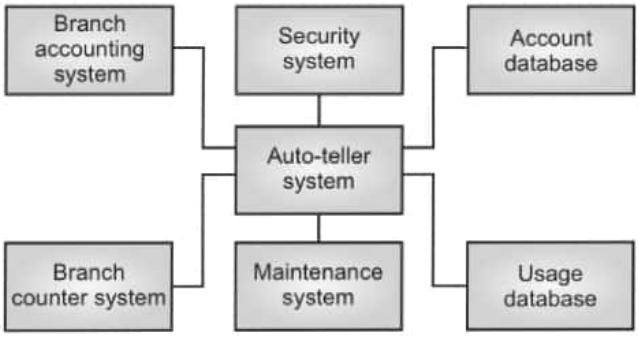


Fig. 3.1 : The context of the ATM system

- There, users range from very young children who can't read through to young adults, their teachers, and school administrators because these groups need different system boundaries; we specified a configuration system that would allow the boundaries to be specified when the system was deployed. The definition of a system boundary is not a value-free judgment. Social and Organizational concerns may mean that the position of a system boundary may be determined by nontechnical factors. For example, a system boundary may be deliberately positioned so that the complete analysis process can be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; and it may be positioned so that the system cost is increased and the system development division must therefore expand to design and implement the system.
- Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.
- Figure 3.1 is a context model that shows the ATM system and the other systems in its environment. You can see that the ATM system is connected to an Security system and a more general customer t record system with which it shares data. The system is also connected to systems branch manager and accounting reporting and customer admissions, and a statistics system that collects information for research.
- Context models normally show that the environment includes several other automated Systems. However, they do not show the types of relationships between the Systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all.

3.3 INTERACTION MODELS

- All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs; interaction between the software being developed and other systems in its environment; or interaction between the components of a software system. User interaction modeling is important as it helps to identify user requirements. Modeling system-to-system interaction highlights the communication problems that may arise. Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- This section discusses two related approaches to interaction modeling:
  1. Use case modeling, which is mostly used to model interactions between systems and external agents (human users or other systems).
  2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.
- Use case models and sequence diagrams present interactions at different levels of detail and so may be used together.
- For example, the details of the interactions involved in a high-level use case may be documented in a sequence diagram.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in

the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.

Use Case Diagram or ATM System:

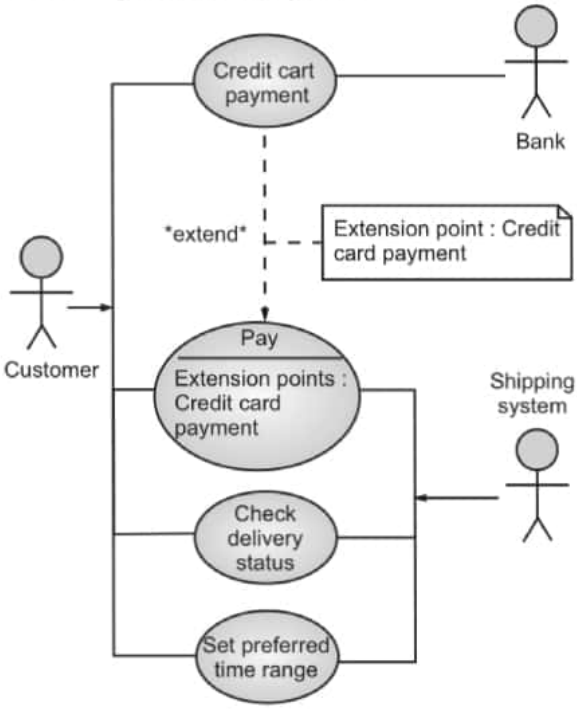


Fig. 3.2 : Use case diagram or ATM system

- Above diagram focuses on interaction between customer, ATM machine and Bank which give scenario based activities to complete transaction.

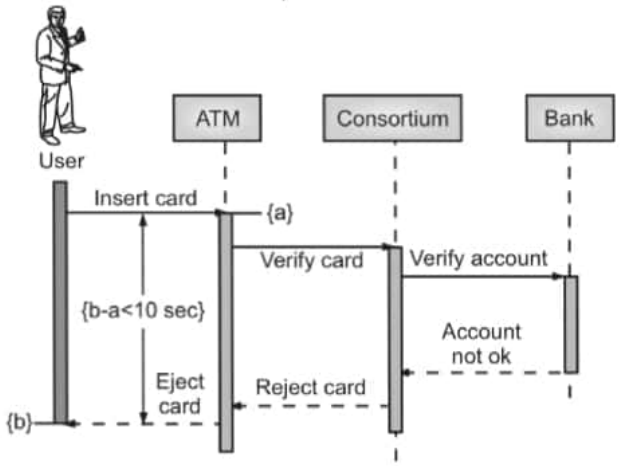


Fig. 3.3 : Sequence diagrams for ATM system

- Above diagram is a Sequence diagrams for ATM system.
- Sequence diagrams, which are used to model interactions between system components although external agents may also be included. Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves. The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be modeled. As space does not allow covering all possibilities here, the focus will be on the basics of this diagram type.
- As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance. Fig. 3.3 is an example of a sequence diagram that illustrates the basics of the notation. This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. Annotated arrows indicate interactions between objects. The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation). You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values.

3.4 STRUCTURAL MODELS

- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- You create structural models of a system when you are discussing and designing the system architecture.
- The class diagram defining the structure of the ATM system is depicted in figure 3.4. It defines several classes: Card, Account, ATM, Transaction, Record, Deposit, and Withdrawal. The class Card owns two Integer attributes number and pin and is associated with the class Account, which owns two Integer attributes number and balance. The described withdrawal functionality is defined by the operation withdraw() of the class ATM, the operation validatePin() of the class Card, as well as the operation make Withdrawal() of the class Account.

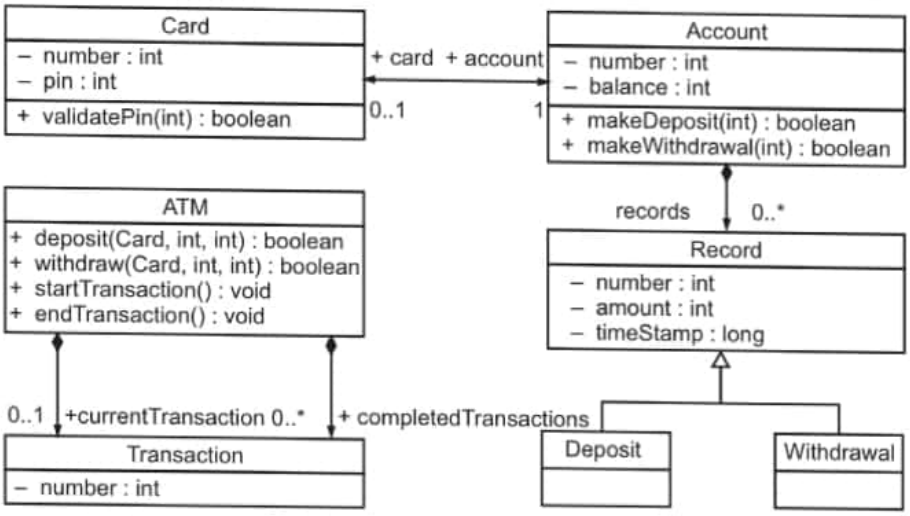


Fig. 3.4 : Class diagram of the ATM system

- The activities defining the behavior of these three operations are depicted in Fig. 3.5 (ATM::withdraw()), Fig. 3.6 (Card::validatePin()), and Fig. 3.7 (Account::make Withdrawal()).
- The activity ATM.Withdraw (Fig. 3.5) defining the behavior of the operation ATM::withdraw() gets as input the card, the pin, and the amount of money to be withdrawn entered by the client. It first starts a new transaction by calling the operation ATM::startTransaction(), then calls the operation Card::validatePin() for validating the pin entered by the client. If the pin is valid, the account associated with the provided card is obtained, and the operation Account::makeWithdrawal() is called to perform the withdrawal of the provided amount of money from the account. Finally, the operation ATM::endTransaction() is called to end the current transaction, and add it to the completed transactions of the ATM.
- The activity Card.ValidatePin (Figure 3.6) defines the behavior of operation Card::validatePin() and checks whether the pin stored on the card matches the provided pin. If this is the case, true is provided as output, otherwise false is provided. If the call of the operation Card::validatePin() provides false as output, the activity ATM. Withdraw also provides false as output indicating an unsuccessful withdrawal. Otherwise, the withdrawal is carried out by calling the operation Account::makeWithdrawal() (action makeWithdrawal) for the account associated with the provided card whose behavior is defined by the activity Account. MakeWithdrawal.

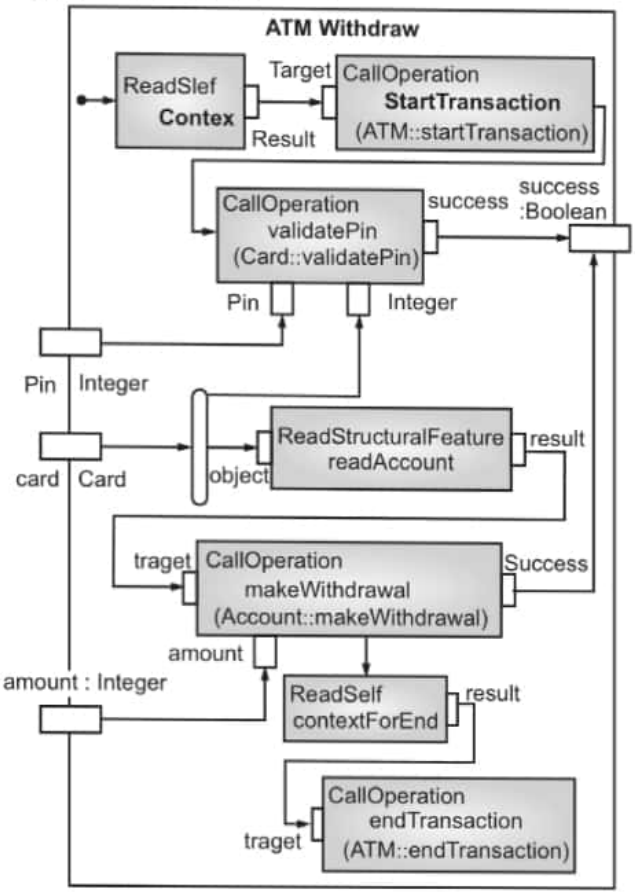


Fig. 3.5 : Activity for operation ATM::withdraw()

- The activity Account. MakeWithdrawal (Fig. 3.7) first checks whether the account's balance exceeds the amount of money to be withdrawn. If this is the case, the balance is reduced by this amount, a new withdrawal record is created (action createNewWithdrawal), and the activity provides true as output indicating the successful update of the account's balance.
- Otherwise false is provided as output. If the activity Account.MakeWithdrawal provides true as output, the activity ATM.Withdraw also provides true as output indicating the successful withdrawal, otherwise false is provided.

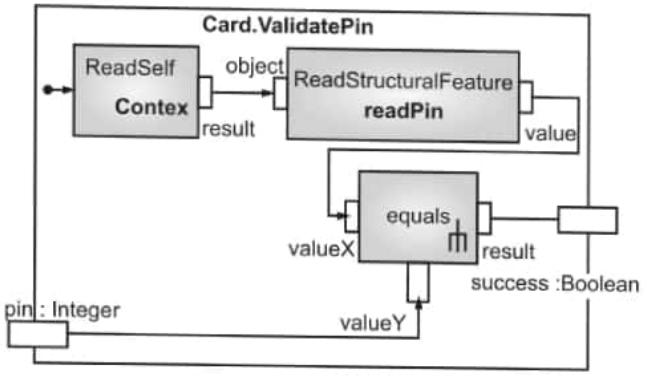


Fig. 3.6. : Activity for operation Card::validatePin()

Fig. 3.7 Class diagram of the ATM system

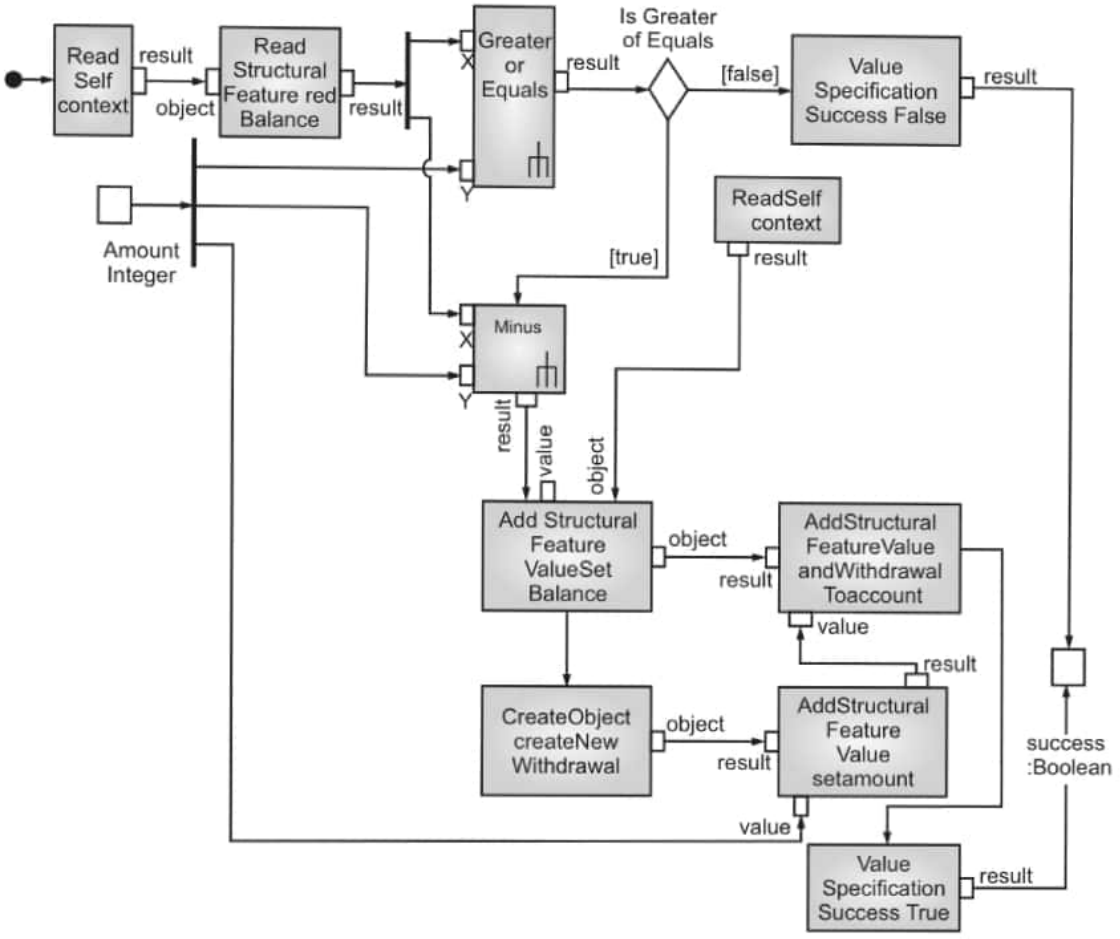


Fig. 3.7 : Activity for operation Account::makeWithdrawal()

3.5 BEHAVIORAL MODELS

- Behavioral models are models of the dynamic behavior of a system as it is executing.
- They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.

- You can think of these stimuli as being of two types:
  1. Data Some data arrives that has to be processed by the system.
  2. Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.
- Behavioral models can provide an unambiguous method for communication between the system and hardware design teams in developing ICs. While the long-stated goals of using HDL models as a practical "executable specification" for a design can remain elusive, behavioral models can provide a common representation between the specifications and assumptions provided in system engineering and the requirements of the hardware design.
- By defining and maintaining key assertions and bounds on the design and maintaining a consistent nomenclature in the development of interfaces from the system down to the partitioned RTL models, behavioral models can be used to both drive and check a system against its original design parameters. Although in many cases behavioral modeling does not present the level of detailed complexity of an RTL model, behavioral modeling presents its own types of modeling challenges.
- Whereas an extensive infrastructure-including libraries, packages and utilities from both electronic design automation and silicon vendors-exists for RTL modeling to facilitate implementing a design in silicon, the support for behavioral modeling is more limited.
- In practice, ATM processing can be extremely complex, both because of the complexities of the processing operations and-equally as critical-because of the coding considerations for efficient implementation to silicon.
- ATM systems typically consist of a number of sizable and complex blocks, including ingress and egress logic, header segmentation and reassembly operations, arbitration and quality-of-service functions, and bus interfaces.

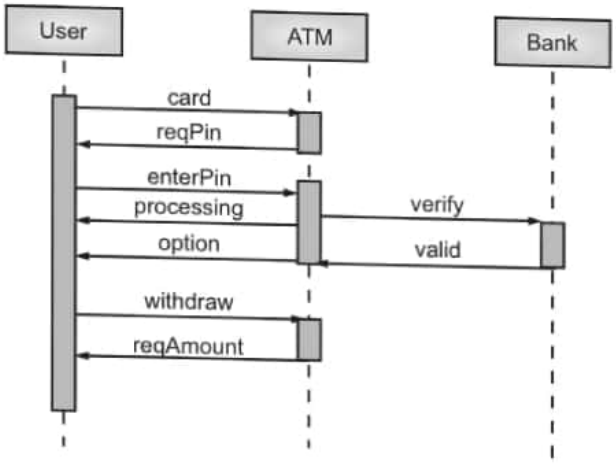


Fig. 3.8 : Sequence diagram of ATM

Example of Hospital Management

- The receptionist logs on to the PRS.
- Two options are available (as shown in the "alt" box). These allow the direct transfer of
- Updated patient information from the Mentcare database to the PRS and the transfer of summary health data from the Mentcare database to the PRS.
- In each case, the receptionist's permissions are checked using the authorization system.
- Personal information may be transferred directly from the user interface object to the PRS.
- Alternatively, a summary record may be created from the database, and that record is then transferred.
- On completion of the transfer, the PRS issues a status message and the user logs off.
- Unless you are using sequence diagrams for code generation or detailed documentation,
- You don't have to include every interaction in these diagrams. If you develop system models early in the development process to support requirements
- Engineering and high-level design, there will be many interactions that depend on implementation decisions

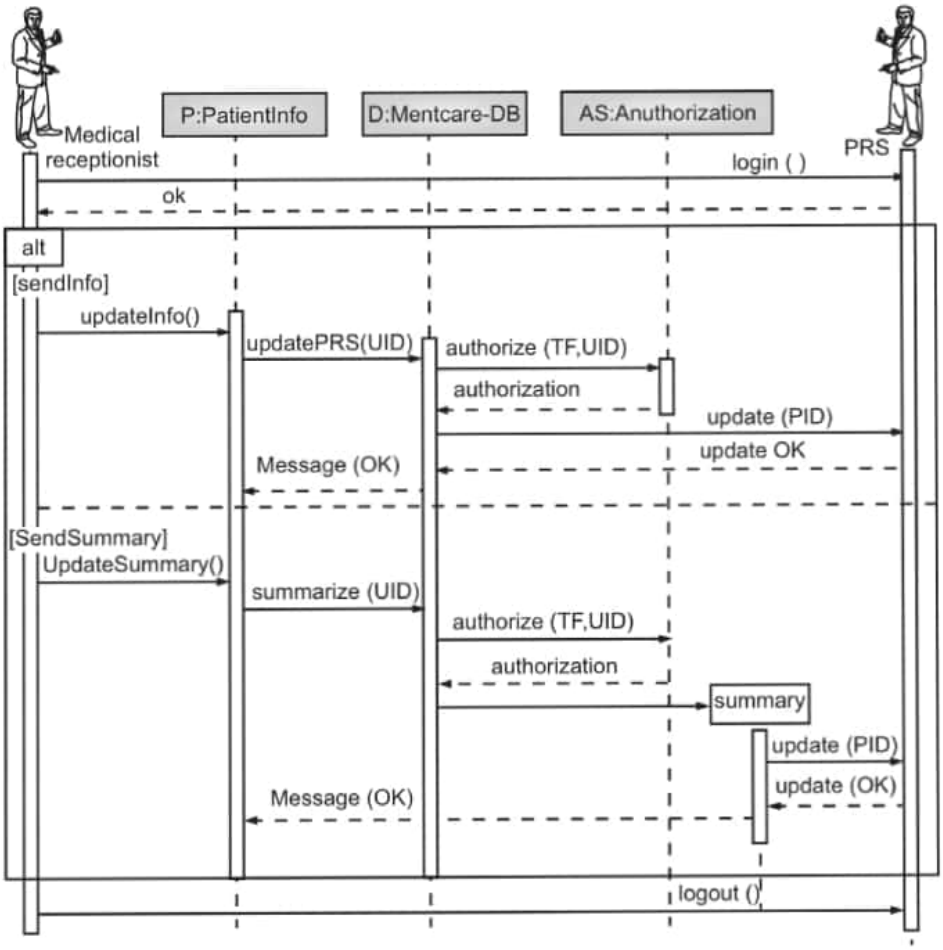


Fig. 3.9 : Sequence diagram for Transfer Data

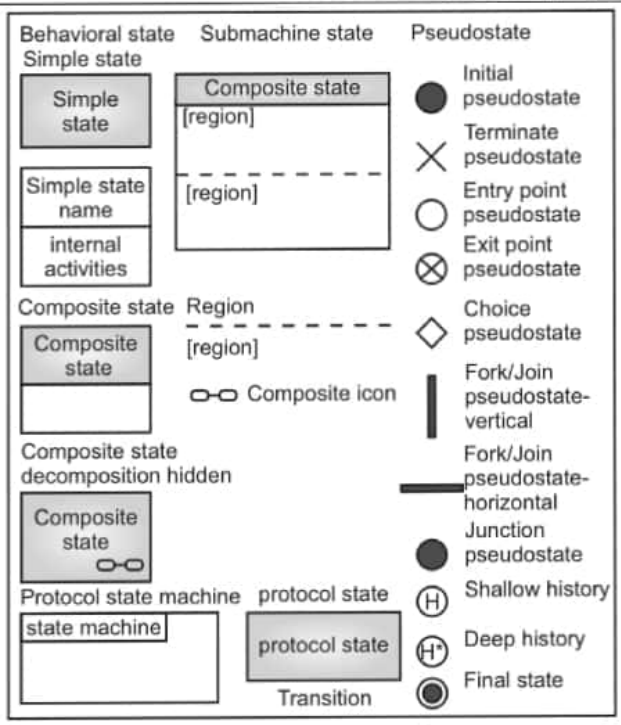


Fig. 3.10 : Behavioral diagram of ATM

3.6 MODEL-DRIVEN ENGINEERING

- Event-driven modeling shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. For example, a system controlling a valve may move from a state "Valve open" to a state "Valve closed" when an operator command (the stimulus) is received.
- Model-Driven Engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
  - ➢ The programs that execute on a hardware/software platform are then generated automatically from the models.
  - ➢ Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with

programming language details or the specifics of execution platforms

- Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.

Pros

- Allows systems to be considered at higher levels of abstraction
- Generating code automatically means that it is cheaper to adapt systems to new platforms.

Cons

- Models for abstraction and not necessarily right for implementation.
- Savings from generating code may be outweighed by the costs of developing translators for new platforms.

Model-Driven Architecture (MDA) was the precursor of more general model-driven engineering

- MDA is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system.
- Models at different levels of abstraction are created. From a high-level, platform independent model, it is possible, in principle, to generate a working program without manual intervention.

MDA Transformation

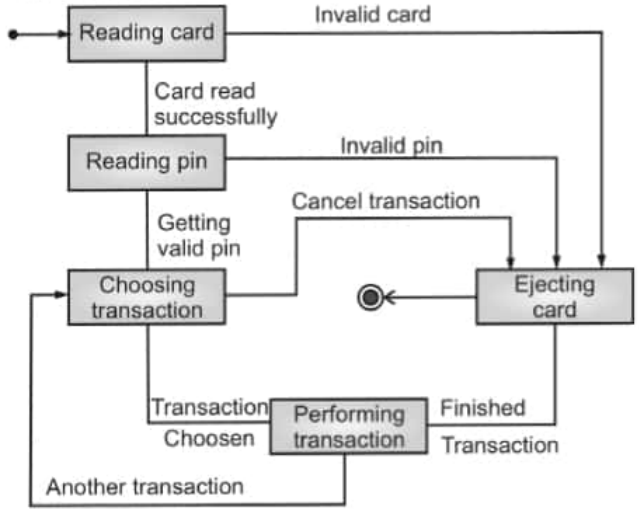


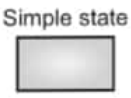
Fig. 3.11 : State transiton diagram for ATM system

- **State Transition Diagram** are also known as Dynamic models. As the name suggests, it is a type of diagram that is used to represent different transition (changing) states of a System. It is generally used to graphically represent all possible transition states a system can have and model such systems. It is very essential and important and right for object-oriented modeling from the beginning.
- The System consists of various states that are being represented using various symbols in the state transition diagram. You can see the symbols and their description given below :

1. **Initial State –**

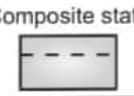
2. **Final State –**

3. **Simple State –**



Advertising

4. **Composite State –**



Type of State	Description
Initial State	In a System, it represents Starting state.
Final State	In a System, it represents Ending state.
Simple State	In a System, it represents a Simple state with no substructure.
Composite State	In a System, it represents a Composite state with two or more parallel or concurrent states out of which only one state will be active at a time and other states will be inactive.

3.7 ARCHITECTURAL DESIGN

3.7.1 Architectural Design Decisions, Views and Patterns

- Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system.

- The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems has been adopted in a number of areas of software engineering. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al. 1995). This prompted the development of other types of patterns, such as patterns for organizational design (Coplien and Harrison 2004), usability patterns (Usability Group 1998), patterns of cooperative interaction (Martin and Sommerville 2004), and configuration management patterns (Berczuk and Appleton 2002). Architectural patterns were proposed in the 1990s under the name "architectural styles" (Shaw and Garlan 1996). A very detailed five-volume series of handbooks on pattern-oriented software architecture was published between 1996 and 2007 (Buschmann et al. 1996; Schmidt et al. 2000; Buschmann, Henney, and Schmidt 2007a, 2007b; Kircher and Jain 2004).

Table 3.1 : The Model-View-Controller (MVC) pattern

Name	MVC (Mode View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g, key presses, mouse clicks, etc) and passes these interactions to the View and the Model. See Fig.3.12
Example	Fig. 3.12 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways, with changes made in one representation shown in all of them.
Disadvantages	May involve additional code and code complexity when the data model and interactions are simple.

- Fig. 3.12 describes the well-known Model-View-Controller pattern. This pattern is the basis of interaction management in many web-based systems and is supported by most language frameworks. The stylized pattern description includes the pattern name, a brief description, a graphical model, and an example of the type of system where the pattern is used. You should also include information about when the pattern should be used and its advantages and disadvantages

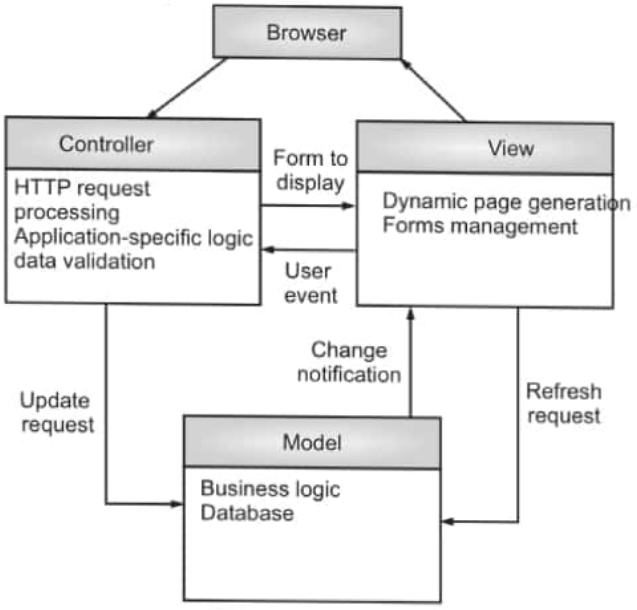


Fig. 3.12

- Web Database application architecture using the MVC pattern

Layered Architecture

- The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. The MVC pattern, shown in Fig. 3.12, separates elements of a system, allowing them to change independently. For example, adding a

new view or changing an existing view can be done without any changes to the underlying data in the model. The Layered Architecture pattern is another way of achieving separation and independence.

- Here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it. This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. If its interface is unchanged, a new layer with extended functionality can replace an existing layer

Name	Layered Architecture
Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system. See Fig. 3.12
Example	A layered model of a digital learning system to support learning of all subjects in schools. Fig. 3.12
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g. authentication) can be provided in each layer to increases the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Repository Architecture

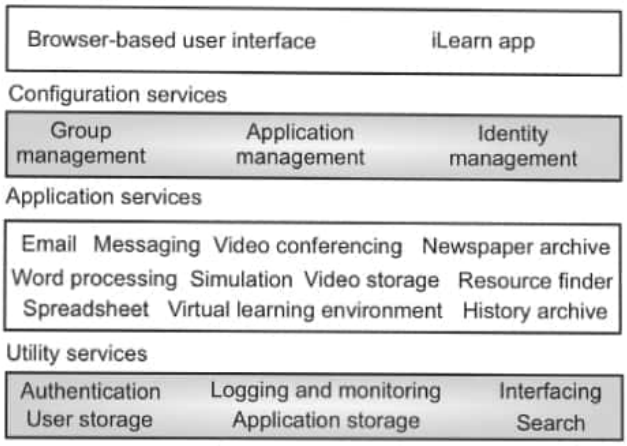


Fig. 3.13

The architecture of the iLearn system

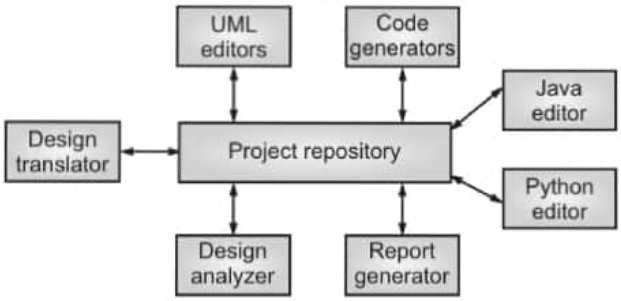


Fig. 3.14: A repository architecture for an IDE

- The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another. Examples of this type of system include command and control systems, management information systems, Computer-Aided Design (CAD) systems, and interactive development environments for software. Fig. 3.14 illustrates a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development. The repository in this case might be a version-controlled environment that keeps track of changes to software and allows rollback to earlier versions. Organizing tools around a repository is an efficient way of sharing large amounts of data. There is no need to transmit data explicitly from one component to another.
- However, components must operate around an agreed repository data model. Inevitably, this is a compromise between the specific needs of each tool, and it maybe

difficult or impossible to integrate new components if their data models do not fit the agreed schema. In practice, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, this involves maintaining multiple copies of data. Keeping these consistent and up to date adds more overhead to the system.

Client–Server Architecture

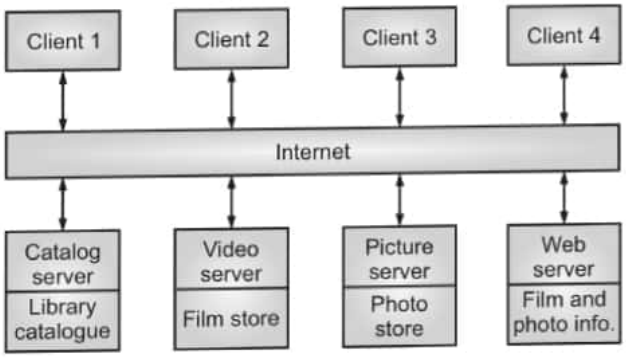


Fig. 3.15 : Client–server architecture for a film library

Pipe and Filter Architecture

- The name "pipe and filter" comes from the original Unix system where it was possible to link processes using "pipes." These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term filter is used because a transformation "filters out" the data it can process from its input data stream.
- Variants of this pattern have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data-processing systems such as billing systems.

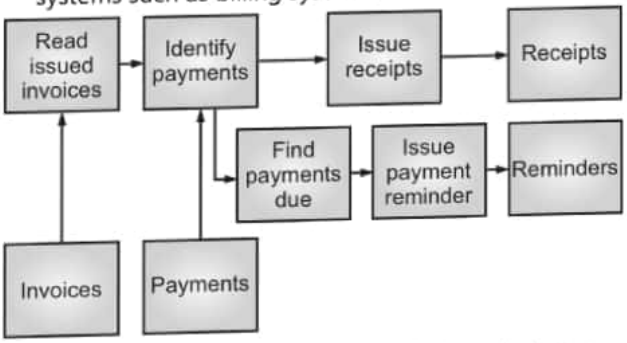


Fig. 3.16: An example of the pipeand filter architecture

Name	Pipe and Filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Fig. 3.16 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overload and may mean that it is impossible to reuse architectural components that use incompatible data structures.

3.8 APPLICATION ARCHITECTURES

- Application systems are intended to meet a business or an organizational need. All businesses have much in common they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector specific applications. Therefore, as well as general business functions, all phone companies need systems to connect and meter calls, manage their network and issue bills to customers. Consequently, the application systems used by these businesses also have much in common.
- These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software systems. Application architectures encapsulate the principal

characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type. The application architecture may be re implemented when developing new systems. However, for many business systems, application architecture reuse is implicit when generic application systems are configured to create a new application. We see this in the widespread use of Enterprise Resource Planning (ERP) systems and off-the-shelf configurable application systems, such as systems for accounting and stock control. These systems have a standard architecture and components. The components are configured and adapted to create a specific business application.

- As a software designer, you can use models of application architectures in a number of ways:
  - ➢ As a starting point for the architectural design process If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. You then specialize this for the specific system that is being developed.
  - ➢ As a design checklist If you have developed an architectural design for an application System, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.
  - ➢ As a way of organizing the work of the development team. The application architectures Identify stable structural features of the system architectures, and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.
  - ➢ As a means of assessing components for reuse If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.
  - ➢ As a vocabulary for talking about applications If you are discussing a specific application or trying to compare applications, then you can use the concepts identified in the generic architecture to talk about these applications.

Some of the Applications are:

Transaction Processing Applications

- Transaction processing systems are designed to process user requests for information from a database, or requests to update a database (Lewis, Bernstein, and Kifer 2003).Technically, a database transaction is part of a sequence of operations and is treated as a single unit (an atomic unit). All of the operations in a transaction have to be completed before the database changes are made permanent. This ensures that failure of operations within a transaction does not lead to inconsistencies in the database.

  Example: To withdraw money from a bank

- An example of a database transaction is a customer request to withdraw money from a bank account using an ATM. This involves checking the customer account balance to see if sufficient funds are available, modifying the balance by the amount withdrawn and sending commands to the ATM to deliver the cash. Until all of these steps have been completed, the transaction is incomplete and the customer accounts database is not changed.

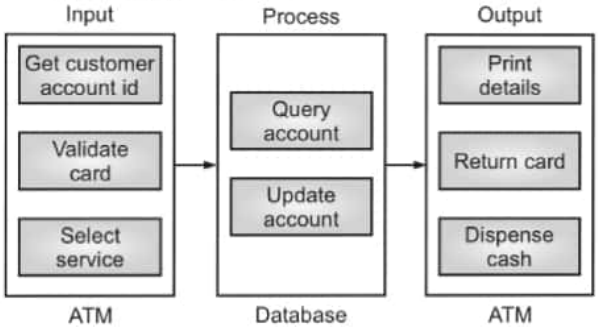


Fig. 3.17: The software architecture of an ATM system

- For example, consider a banking system that allows customers to query their accounts and withdraw cash from an ATM. The system is composed of two

cooperating software components the ATM software and the account processing software in the bank's database server. The input and output components are implemented as software in the ATM, and the processing component is part of the bank's database server.

Information Systems

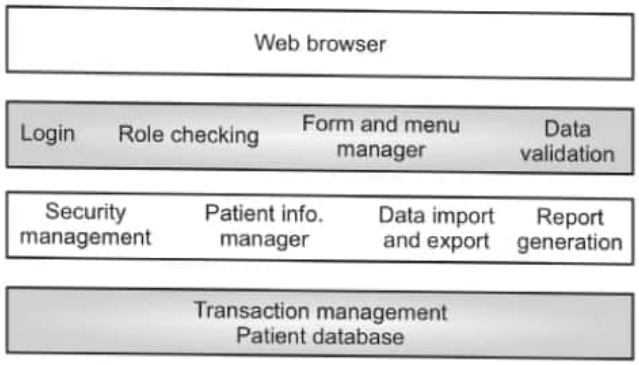


Fig. 3.18: Layered information system architecture

- We have added detail to each layer in the model by identifying the components that support user communications and information retrieval and access:
  - ➢ The top layer is a browser-based user interface.
  - ➢ The second layer provides the user interface functionality that is delivered through the web browser. It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role. This layer includes form and menu management components that present information to users, and data validation components that check information consistency.
  - ➢ The third layer implements the functionality of the system and provides Components that implement system security, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.
  - ➢ Finally, the lowest layer, which is built using a commercial database management system, provides transaction management and persistent data storage.

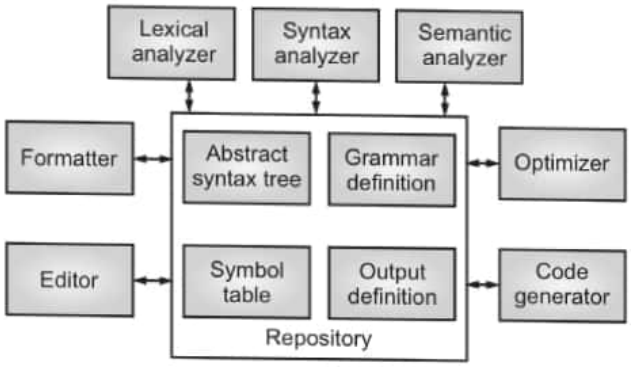


Fig. 3.19: Repository architecture for a language processing system

- Programming language compilers that are part of a more general programming environment have a generic architecture (Fig. 3.19) that includes the following components:
  - ➢ A lexical analyzer, which takes input language tokens and converts them into an internal form.
  - ➢ A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
  - ➢ A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.
  - ➢ A syntax tree, which is an internal structure representing the program being compiled.
  - ➢ A semantic analyzer, which uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
  - ➢ A code generator, which "walks" the syntax tree and generates abstract machine code.

EXERCISE

1. Explain Context models,
2. Explain interaction model.
3. Explain Context models.
4. Draw Use case diagram for ATM system.
5. Draw Sequence diagrams for ATM system.

6. Explain Model-Driven Engineering (MDE).
7. Draw State transiton diagram for ATM system.
8. Draw the software architecture of an ATM system.
9. Explain Layered information system architecture.
10. Explain Client–Server Architecture.
11. Explain in detail.
    (a) Layered Architecture.
    (b) Repository Architecture
    (c) Pipe and Filter Architecture
    (d) Client–Server Architecture

✠ ✠ ✠

DESIGN AND IMPLEMENTATION

4.1 OBJECT-ORIENTED DESIGN USING THE UML

- An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions. Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understand ability, and hence the maintainability, of the design.
- To develop a system design from concept to detailed, object-oriented design, you need to:
  - ➢ Understand and define the context and the external interactions with the system.
  - ➢ Design the system architecture.
  - ➢ Identify the principal objects in the system.
  - ➢ Develop design models.
  - ➢ Specify interfaces.
- In the use case diagram, there are two actors named student and a teacher. There are a total of five use cases that represent the specific functionality of a student management system. Each actor interacts with a particular use case. A student actor can check attendance, timetable as well as test marks on the application or a system. This actor can perform only these interactions with the system even though other use cases are remaining in the system.

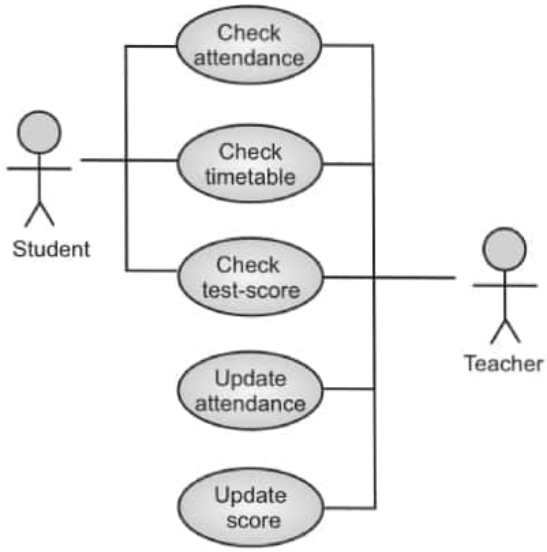


Fig. 4.1 : Student Management System

- It is not necessary that each actor should interact with all the use cases, but it can happen.
- The second actor named teacher can interact with all the functionalities or use cases of the system. This actor can also update the attendance of a student and marks of the student. These interactions of both student and teacher actor together sums up the entire student management application.
- Each of these use cases should be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do.
  - ➢ **System** Weather station
  - ➢ **Use Case** Report weather
  - ➢ **Actors** Weather information system, Weather station
  - ➢ **Data** The weather station sends a summary of the weather data that has been collected from the Instruments in the collection period to the weather information system. The data sent are the

maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.

- ➢ **Stimulus** The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
- ➢ **Response** The summarized data is sent to the weather information system.
- ➢ **Comments** Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future

4.1.1 System Context and Interactions

- The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. Understanding the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems. In this case, you need to decide how functionality is distributed between the control system for all of the weather stations and the embedded software in the weather station itself.
- System context models and interaction models present complementary views of the relationships between a system and its environment:
  - ➢ A system context model is a structural model that demonstrates the other systems in the Environment of the system being developed.
  - ➢ An interaction model is a dynamic model that shows how the system interacts with its Environment as it is used.
- The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations.
- Fig. 4.1 shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system. When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail. One way to do this is to use a use case model.

4.1.2 Object Class Identification

- By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing. As your understanding of the design develops, you refine these ideas about the system objects. The use case description helps to identify objects and operations in the system. As object-oriented design evolved in the 1980s, various ways of identifying object classes in object-oriented systems were suggested:
  - ➢ Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs (Abbott 1983).
  - ➢ Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations such as offices, organizational units such as companies, and so on (Wirfs-Brock, Wilkerson, and Weiner 1990).
  - ➢ Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations (Beck and Cunningham 1989).

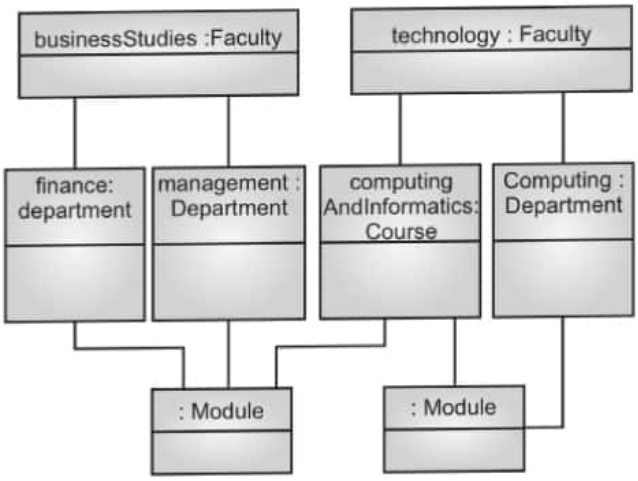


Fig. 4.2 : Student Management System

- The object diagram shown above is designed to highlight the fact that a single course (in this case "Computing and Informatics") is administered by the Faculty of Technology, and includes a number of modules administered by the Computing department (as you might expect, given the title of the course). It also tells us that the course includes modules administered by both the Management department and the Finance department, both of which are in turn administered by the Faculty of Business Studies.
- Object diagrams can be used to highlight all kinds of situations that can arise at some moment in the life of an information system, at an arbitrary level of detail (the level of detail shown on an object diagram should only be as much as is necessary to make the point we are trying to make). They are useful for helping stakeholders to understand a class diagram, or for showing a client how part of the system is supposed to work in a given scenario. They are *not* as a rule used by system designers or programmers to design an application or write program code (class diagrams are far better suited for that purpose).
- An object diagram can sometimes be used to test the accuracy of a corresponding class diagram, or to show how an instantiated object relates to its class definition. Although it is not common for classes and objects to appear on the same diagram, sometimes part of a larger class diagram may be abstracted onto a separate diagram, together with an object diagram that provides a concrete example of the relationship between the classes involved.

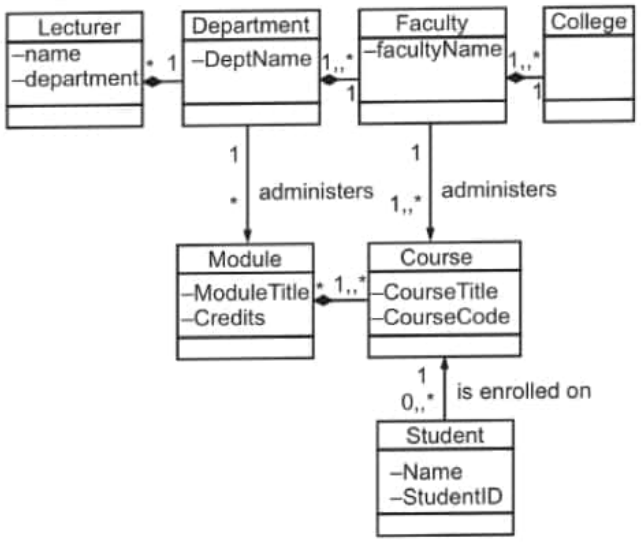


Fig. 4.3 : Faculty System (Object)

- Identify the general object classes in the system
- In order to illustrate a situation in which an object diagram might be used to clarify a situation that a class diagrams alone may not adequately explain, consider the class diagram below, which describes a college information system. It might be useful to first examine just what this diagram *does* tell us about the system, and then consider what it might not tell us.
- Above diagram tells us the following things:
  - ➢ The college is an aggregation of one or more faculties, and a faculty can only belong to one college.
  - ➢ Each faculty is an aggregation of one or more departments, and a department may only belong to one faculty.
  - ➢ Each department is an aggregation of a number of lecturers, and a lecturer may only belong to one department.
  - ➢ Each faculty administers one or more courses, and a course may only be administered by one faculty.
  - ➢ Each department administers one or more modules, and a module may only be administered by one department.
  - ➢ Each course is an aggregation of one or more modules, and a module may belong to one or more courses.
  - ➢ Each course may have zero or more students enrolled on it, but a student must be enrolled on one (and only one) course.

4.1.3 Design Models

- Design or system models, show the objects or object classes in a system. They also show the associations and relationships between these entities.
- These models are the bridge between the system requirements and the implementation of a system. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements.
- The level of detail that you need in a design model depends on the design process used. Where there are close links between requirements engineers, designers and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. Similarly, if agile development is used, outline design models on a whiteboard may be all that is required.
- If a plan-based development process is used, you may need more detailed models. When the links between requirements engineers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then precise design descriptions are needed for Communication. Detailed models, derived from the high-level abstract models, are used so that all team members have a common understanding of the design.
- An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. A sequential data-processing system is quite different from an embedded real-time system, so you need to use different types of design models.
- When you use the UML to develop a design, you should develop two kinds of design model:

1. Structural Models :

- It describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.

2. Dynamic Models:

- It describes the dynamic structure of the system and show the expected runtime interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions.
- Three UML model types are particularly useful for adding detail to use case and architectural models:
  - ➢ Sub system models, which show logical groupings of objects into coherent sub systems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are structural models.
  - ➢ Sequence models, which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
  - ➢ State machine models, which show how objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.
- A sub system model is a useful static model that shows how a design is organized into logically related groups of objects. To present the sub systems in the weather mapping system. As well as subsystem models, you may also design detailed object models, showing the objects in the systems and their associations (inheritance, generalization, aggregation, etc.).
- Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place. When documenting a design, you should produce a sequence model for each significant interaction. If you have developed a use case model, then there should be a sequence model for each use case that you have identified.

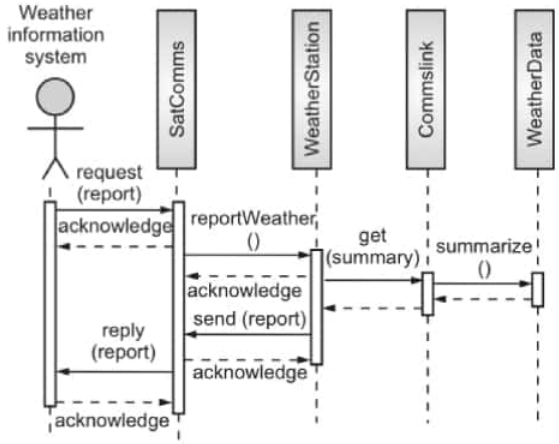


Fig. 4.4 : Sequence diagram for Weather Information System

- Fig. 4.4 is an example of a sequence model, shown as a UML sequence diagram.
- This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:
  - ➢ The SatComms object receives a request from the weather information system to collect a weather report from a weatherStation. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.
  - ➢ SatComms sends a message to Weather Station, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.
  - ➢ WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.
  - ➢ Commslink calls the summarize method in the object Weather Data and waits for a reply.
  - ➢ The weather data summary is computed and returned to WeatherStation via the Commslink object.
  - ➢ Weather Station then calls the SatComms object to transmit the summarized data to the weather information system, through the satellite communications system.

Interface Specification

- An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and subsystems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.
- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section, and the UML stereotype «interface» should be included in the name part. The semantics of the interface may be defined using the Object Constraint Language (OCL).

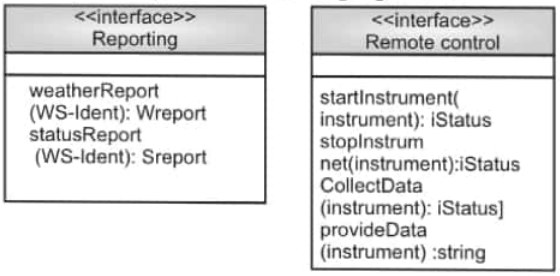


Fig. 4.5 : WeatherSation Interfaces

- Fig. 4.5 shows two interfaces that may be defined for the weatherStation. The left hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports. These map directly to operations in the Weather Station object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object. In this case, the individual operations are encoded in the command string associated with the remote Control method, shown in Fig. 4.5.

4.2 DESIGN PATTERNS

- Design patterns were derived from ideas put forward by Christopher Alexander(Alexander 1979), who suggested that there were certain common patterns of building design that were inherently pleasing and effective.

Definition:

- "The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different Way;"
- The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.
- Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.
- Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design.
- This is particularly true for the best known design patterns that were originally described by the "Gang of Four" .
- **Pattern Name:** Observer
- **Description**: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
- **Problem Description**: This pattern may be used in situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
- **Solution Description:** This involves two abstract objects, Subject and Observer, and two concrete objects, Concrete Subject and Concrete Object, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in Concrete Subject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.
- **Consequences:** The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

The Gang of Four defined the four essential elements of design patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences the results and trade-offs of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

   Gamma and his co-authors break down the problem description into motivation (a description of why the patter is useful) and applicability (a description of situations in which the pattern may be used). Under the description of the solution, they describe the pattern structure, participants, collaborations, and implementation.

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied. Examples of such problems as per GOF is as Below

- Tell several objects that the state of some other object has changed (Observer pattern).
- Tidy up the interfaces to a number of related objects that have often been developed Incrementally (Façade pattern).

- Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
- Allow for the possibility of extending the functionality of an existing class at runtime (Decorator pattern).

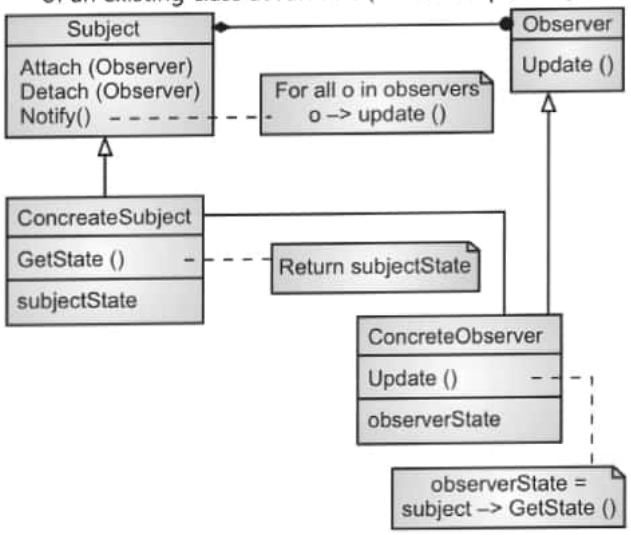


Fig. 4.6 : A UML model of the Observer pattern

4.3 IMPLEMENTATION ISSUES

- Software development must follow the process from the initial requirements of the system through to maintenance and management of the deployed system.
- A critical stage of this process is, of course, system implementation, where you create an executable version of the software. Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

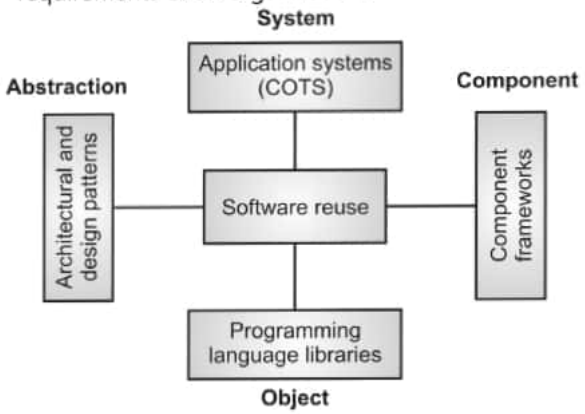


Fig. 4.7 : Reuse of Software

These are:

1. **Reuse :** Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
2. **Configuration Management :** During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.
3. **Host-Target Development :** Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type, but often they are completely different.

Reuse

- In the era of 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Consequently, an approach to development based on the reuse of existing software is now the norm for many types of system development. A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering.
- Software reuse is possible at a number of different levels, as shown in Fig. 4.7
  - ➢ The abstraction level At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software.
  - ➢ The object level At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process

email messages in a Java program, you may use objects and methods from a Java Mail library.

- ➢ The component level Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.
- ➢ The system level at this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic application systems are adapted and reused.

Configuration Management

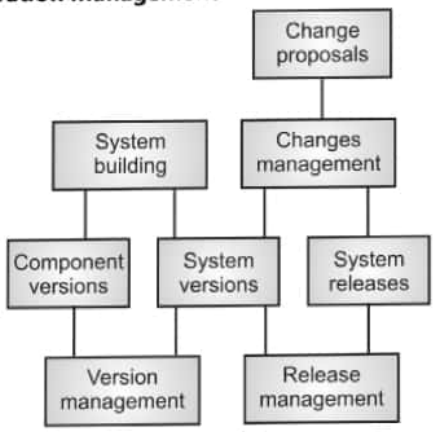


Fig. 4.8 : Configuration of system

- In software development, change happens all the time, so change management is absolutely essential. When several people are involved in developing a software system, you have to make sure that team members don't interfere with each other's work. That is, if two people are working on a component, their changes have to be coordinated. Otherwise, one programmer may make changes and overwrite the other's work. You also have to ensure that everyone can access the most up-to-date versions of software components; otherwise developers may redo work that has already been done. When something goes wrong with a new version of a system, you have to be able to go back to a working version of the system or component.
- Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to 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, and compile and link components to create a system. As shown in Fig. 4.8 there are four fundamental configuration management activities:

1. **Version Management,** where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer from overwriting code that has been submitted to the system by someone else.
2. **System Integration**, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
3. **Problem Tracking**, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.
4. **Release Management,** where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

4.4 OPEN-SOURCE DEVELOPMENT

- Open-source development is 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 (Raymond 2001). Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code

should not be proprietary but rather should always be available for users to examine and modify as they wish. There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code.

- Open-source software extended this idea by using the Internet to recruit a much larger Population of volunteer developers. Many of them are also users of the code. In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality. However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.
- Open-source software is the backbone of the Internet and software engineering. The Linux operating system is the most widely used server system, as is the open-source
- Apache web server. Other important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system. The Android operating system is installed on millions of mobile devices. Major players in the computer industry such as IBM and Oracle, support the open-source movement and base their software on open-source products. Thousands of other, lesser-known open-source systems and components may also be used.
- For a company involved in software development, there are two open-source issues that have to be considered:
  1. Should the product that is being developed make use of open-source components?
  2. Should an open-source approach be used for its own software development?
- The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.
- If you are developing a software product for sale, then time to market and reduced costs are critical. If you are developing software in a domain in which there are high-quality open-source systems available, you can save time and money by using these systems.
- However, if you are developing software to a specific set of organizational requirements, then using open-source components may not be an option. You may have to integrate your software with existing systems that are incompatible with available Publishing the source code of a system does not mean that people from the wider community will necessarily help with its development. Most successful open-source products have been platform products rather than application systems. There are a limited number of developers who might be interested in specialized application systems. Making a software system open source does not guarantee community involvement.
- There are thousands of open-source projects on Sourceforge and GitHub that have only a handful of downloads. However, if users of your software have concerns about its availability in future, making the software open source means that they can take their own copy and so be reassured that they will not lose access to it.

4.4.1 Open-Source Licensing

- Most open-source licenses (Chapman 2010) are variants of one of three general models:
  1. The GNU General Public License (GPL). This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.
  2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open-source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.
  3. The Berkley Standard Distribution (BSD) License. This is a nonreciprocal license, which means you are not obliged to re-publish any changes or modifications made to you use open-source components, you must acknowledge the original creator of the code. The MIT license is a variant of the BSD license with similar conditions. Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source. If you are trying to sell your software, you may wish to keep

it secret. This means that you may wish to avoid using GPL-licensed open source software in its development.

- If you are building software that runs on an open-source platform but that does not reuse open-source components, then licenses are not a problem. However, if you embed open-source software in your software, you need processes and databases to keep track of what's been used and their license conditions. (Bayersdorfer 2007) suggests that companies managing projects that use open source should:
  - ➢ Establish a system for maintaining information about open-source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change, so you need to know the conditions that you have agreed to.
  - ➢ Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.
  - ➢ Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.
  - ➢ Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open-source licensing.
  - ➢ Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.
  - ➢ Participate in the open-source community. If you rely on open-source products, you should participate in the community and help support their development.

4.4.2 Open Source Licenses by Category

- **License Index**
  - ➢ License Approval Process
  - ➢ License Information
  - ➢ Origins and definitions of categories from the License Proliferation Committee report
- In the lists below, a parenthesized expression following a license name is its SPDX short identifier, if one exists, except for two items in the first list (GNU General Public License and GNU Lesser General Public License). For these, the parenthesized expressions ("GPL" and "LGPL" respectively) are the common non-version-specific names of these licenses today (note also that the full name of the first version (2.0) of the LGPL is the GNU Library General Public License). There is no non-version-specific SPDX short identifier for the GPL and LGPL.
- Licenses that are "popular and widely-used or with strong communities"
- The below list is based on publicly available statistics obtained at the time of the Report of License Proliferation Committee.
  - ➢ Apache License 2.0 (Apache-2.0)
  - ➢ 3-clause BSD license (BSD-3-Clause)
  - ➢ 2-clause BSD license (BSD-2-Clause)
  - ➢ GNU General Public License (GPL)
  - ➢ GNU Lesser General Public License (LGPL)
  - ➢ MIT license (MIT)
  - ➢ Mozilla Public License 2.0 (MPL-2.0)
  - ➢ Common Development and Distribution License 1.0 (CDDL-1.0)
  - ➢ Eclipse Public License 2.0 (EPL-2.0)

International Licenses

- ➢ CeCILL License 2.1
- ➢ European Union Public License (EUPL-1.2)
- ➢ LicenceLibre du Québec – Permissive (LiLiQ-P) version 1.1 (LiLiQ-P-1.1)

- LicenceLibre du Québec – Réciprocité (LiLiQ-R) version 1.1 (LiLiQ-R-1.1)
- LicenceLibre du Québec – Réciprocité forte (LiLiQ-R+) version 1.1 (LiLiQ-Rplus-1.1)
- Mulan Permissive Software License v2 (MulanPSL - 2.0)

Special Purpose Licenses

- Certain licensors, such as schools and the US government, have specialized concerns, such as specialized rules for government copyrights. Licenses that were identified by the License Proliferation Committee as meeting a special need were placed in this group.
  - BSD+Patent (BSD-2-Clause-Patent)
  - Educational Community License, Version 2.0 (ECL-2.0)
  - IPA Font License (IPA)
  - Lawrence Berkeley National Labs BSD Variant License (BSD-3-Clause-LBNL)
  - NASA Open Source Agreement 1.3 (NASA-1.3)
  - OSET Public License version 2.1 (OSET-PL-2.1)
  - SIL Open Font License 1.1 (OFL-1.1)
  - Unicode License Agreement - Data Files and Software
  - The Unlicense (Unlicense)
  - Upstream Compatibility License v1.0 (UCL-1.0)

EXERCISE

1. Assume that the Library management system is being developed using an object-oriented approach. Draw a use case diagram showing at least six possible use cases for this system.
2. Using the UML graphical notation for object classes, design the following object classes, identifying attributes and operations. Use your own experience to decide on the attributes and operations that should be associated with these objects.

   A messaging system on a mobile (cell) phone or tablet.

   A printer for a personal computer.

   A personal music system.

   A bank account.

   A library catalogue.
3. Develop the design of the Hospital Management system to show the interaction between the data collection subsystem and the instruments that collect patient data. Use sequence diagrams to show this Interaction.
4. Identify possible objects in the following systems and develop an object-oriented design for them. You may make any reasonable assumptions about the systems when deriving the design. A group diary and time management system is intended to support the timetabling of meetings and appointments across a group of co-workers. When an appointment is to be made that involves a number of people, the system finds a common slot in each of their diaries and arranges the appointment for that time. If no common slots are available, it interacts with the user to rearrange his or her personal diary to make room for the appointment.
5. A filling station (gas station) is to be set up for fully automated operation. Drivers swipe their credit card through a reader connected to the pump; the card is verified by communication with a credit company computer, and a fuel limit is established. The driver may then take the fuel required. When fuel delivery is complete and the pump hose is returned to its holster, the driver's credit card account is debited with the cost of the fuel taken. The credit card is returned after debiting. If the card is invalid, the pump returns it before fuel is dispensed.
6. Draw a sequence diagram showing the interactions of objects in a group diary system when a group of people are arranging a meeting.
7. Draw a UML state diagram showing the possible state changes in either the group diary or the filling station system.
8. When code is integrated into a larger system, problems may surface. Explain how Configuration management can be useful when handling such problems.

9. A small company has developed a specialized software product that it configures specially for each customer. New customers usually have specific requirements to be incorporated into their system, and they pay for these to be developed and integrated with the product. The software company has an opportunity to bid for a new contract, which would more than double its customer base. The new customer wishes to have some involvement in the configuration of the system. Explain why, in these circumstances, it might be a good idea for the company owning the software to make it open source.

✠ ✠ ✠

TESTING

5.1 SOFTWARE TESTING

5.1.1 Introduction

- **Software Testing** is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free. It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.
- **Software Testing** is Important because if there are any bugs or errors in the software, it can be identified early and can be solved before delivery of the software product. Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.
  - ➢ Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
  - ➢ When you test software, you execute a program using artificial data.
  - ➢ You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
  - ➢ Testing can reveal the presence of errors NOT their absence.
  - ➢ Testing is part of a more general verification and validation process, which also includes static validation techniques.

Program Testing Goals

- To demonstrate to the developer and the customer that the software meets its requirements.
  - ➢ For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - ➢ Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.
- The first goal leads to validation testing
  - ➢ You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- The second goal leads to defect testing
  - ➢ The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

Benefits of Software Testing:

- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product Quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

5.1.2 Validation and Defect Testing

Validation Testing

- To demonstrate to the developer and the system customer that the software meets its requirements
- A successful test shows that the system operates as intended.

Defect Testing

- To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

An Input-Output Model of Program Testing

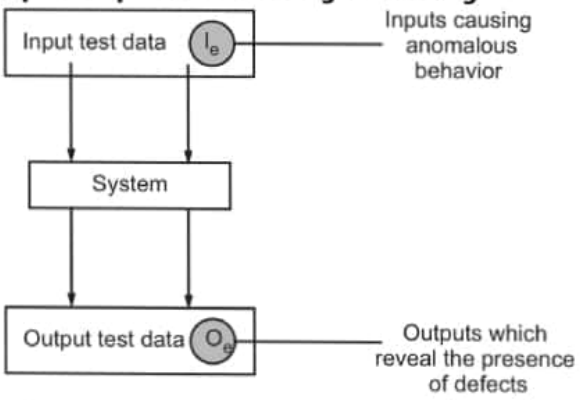


Fig. 5.1 : An input-output model of program testing

Verification V/s Validation (V&V)

- **Verification:** "Are we building the product right".
  The software should conform to its specification.
- **Validation:** "Are we building the right product".
  The software should do what the user really requires.
- Aim of V & V is to establish confidence that the system is 'fit for purpose'.
- Depends on system's purpose, user expectations and marketing environment
  - ➢ Software purpose
    The level of confidence depends on how critical the software is to an organization.
  - ➢ User expectations
    Users may have low expectations of certain kinds of software.
  - ➢ Marketing environment
    Getting a product to market early may be more important than finding defects in the program.

5.1.3 Inspections and Testing

- **Software Inspections :** Concerned with analysis of the static system representation to discover problems (static verification)
  - ➢ May be supplement by tool-based document and code analysis
- **Software Testing :** Concerned with exercising and observing product behavior (dynamic verification)
  - ➢ The system is executed with test data and its operational behavior is observed.

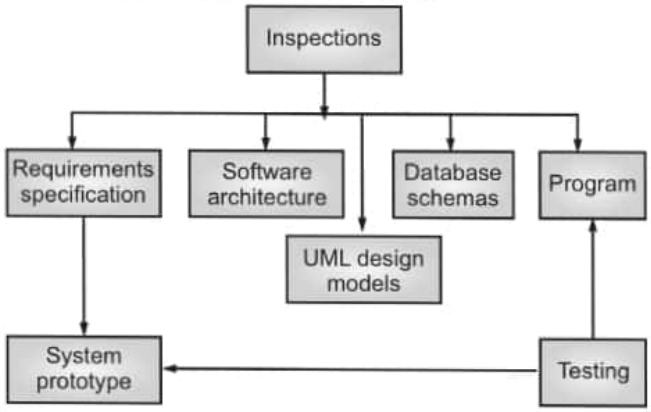


Fig. 5.2 : Inspections and testing

Software Inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Advantages of Inspections

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

Inspections and Testing

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

5.1.4 A Model of the Software Testing Process

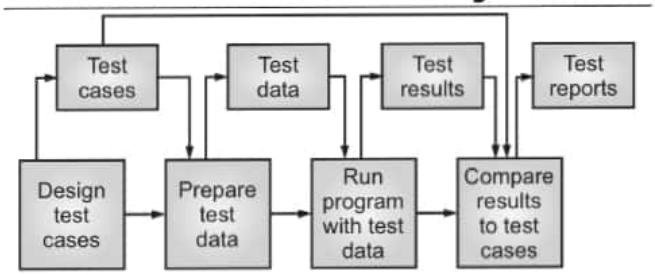


Fig. 5.3 : A model of the software testing process

- Above is an abstract model of the traditional testing process, as used in plan driven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested. Test data are the inputs that have been devised to test a system. Test data can sometimes be generated automatically, but automatic test case generation is impossible.
- People who understand what the system is supposed to do must be involved to specify the expected test results. However, test execution can be automated. The test results are automatically compared with the predicted results, so there is no need for a person to look for errors and anomalies in the test run. Typically, a commercial software system has to go through three stages of testing.

Stages of Testing

- **Development Testing:** where the system is tested during development to discover bugs and defects.
- **Release Testing:** where a separate testing team tests a complete version of the system before it is released to users.
- **User Testing**: where users or potential users of a system test the system in their own environment.

5.2 DEVELOPMENT TESTING

- Development testing includes all testing activities that are carried out by the team developing the system.
- **Unit Testing:** where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
- **Component Testing:** where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
- **System Testing:** where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

5.2.1 Unit Testing

- Unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Units may be:
  - ➢ Individual functions or methods within an object
  - ➢ Object classes with several attributes and methods
  - ➢ Composite components with defined interfaces used to access their functionality.

Object Class Testing

- Complete test coverage of a class involves
  - ➢ Testing all operations associated with an object
  - ➢ Setting and interrogating all object attributes
  - ➢ Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localized

Example: Weather station testing

WeatherStation
identifier
reporWeather () reportStatus ( ) powerSave (instruments) remoteControl (commands) restart (instruments) shutdown (instruments)

Fig. 5.4 : The weather station object interface

- Need to define test cases for report Weather, calibrate, test, startup and shutdown.

- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- For example:
  - ➢ Shutdown -> Running->Shutdown
  - ➢ Configuring-> Running->Testing -> Transmitting ->Running->
  - ➢ Running-> Collecting-> Running-> Summarizing ->Transmitting->Running

Automated Testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.

Automated Test Components

- A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
- A call part, where you call the object or method to be tested.
- An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

Unit Test Effectiveness

- The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- If there are defects in the component, these should be revealed by test cases.
- This leads to two types of unit test case:
  1. The first of these should reflect normal operation of a program and should show that the component works as expected.
  2. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

5.2.2 Testing Strategies

- Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:
  1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
  2. If there are defects in the component, these should be revealed by test cases.

Two strategies that can be effective in helping you choose test cases are:

- Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - ➢ You should choose tests from within each of these groups.
- Guideline-based testing, where you use testing guidelines to choose test cases.
  - ➢ These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Partition Testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

Equivalence Partitioning

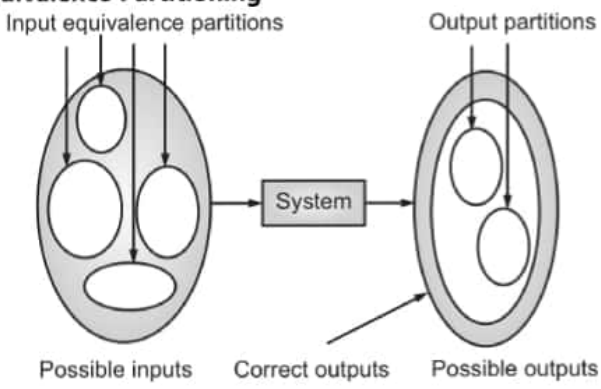


Fig. 5.5 : Equivalence partitioning

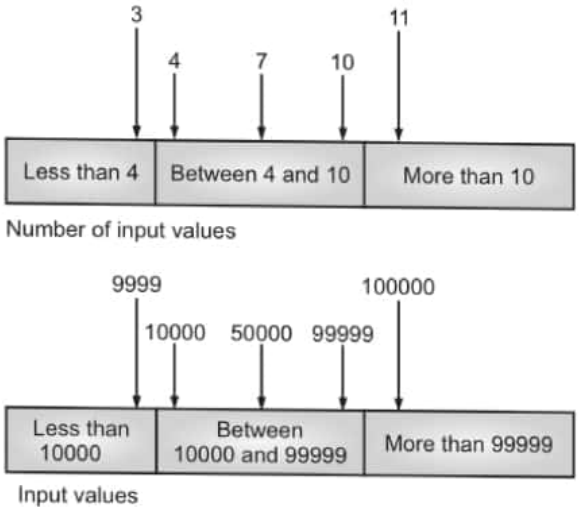


Fig. 5.6 : Equivalence partitions

Testing Guidelines (Sequences)

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.

General Testing Guidelines

- 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.

5.2.3 Component Testing

- Software components are often composite components that are made up of several interacting objects.

  For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
- You can assume that unit tests on the individual objects within the component have been completed.

Interface Testing

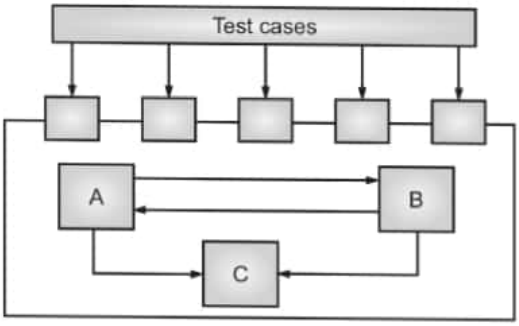


Fig. 5.7: Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types
  - ➢ Parameter interfaces Data passed from one method or procedure to another.
  - ➢ Shared memory interfaces Block of memory is shared between procedures or functions.
  - ➢ Procedural interfaces Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - ➢ Message passing interfaces Sub-systems request services from other sub-systems

Interface Errors

- Interface misuse
  - ➢ A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding
  - ➢ A calling component embeds assumptions about the behaviour of the called component which are incorrect.

- Timing errors
  - ➢ The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface Testing Guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

5.2.4 System Testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the emergent behaviour of a system.

System and Component Testing

- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
  - ➢ In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Use-Case Testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.

Collect Weather Data Sequence Chart

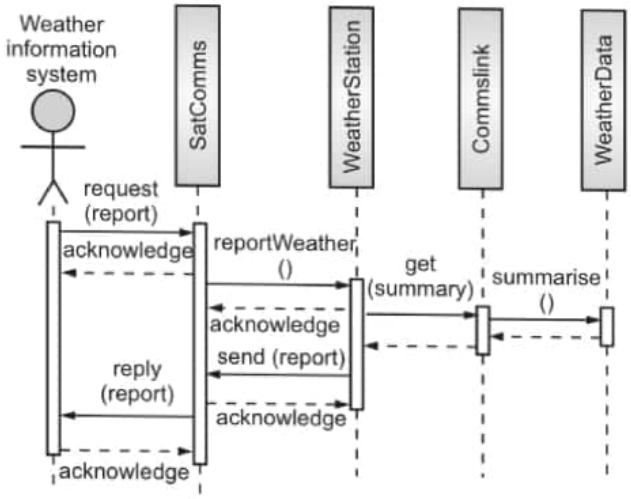


Fig. 5.8 : Collect weather data sequence chart

Testing Policies

- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  - ➢ All system functions that are accessed through menus should be tested.
  - ➢ Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - ➢ Where user input is provided, all functions must be tested with both correct and incorrect input.

5.3 TEST-DRIVEN DEVELOPMENT

- Test-Driven Development (TDD) is an evolutionary approach to development which combines Test-First Development (TFD) where you write a test before you write just enough production code to fulfill that test and refactoring. , it's one way to think through your requirements or design before you write your functional code.

- (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests.
- In TDD three activities are strongly interlinked: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).
- It can be briefly described by the following set of rules:
  - ➢ Write a "single" unit test describing an aspect of the program.
  - ➢ Run the test, which should fail because the program lacks that feature.
  - ➢ Write "just enough" code, the simplest possible, to make the test pass.
  - ➢ "Refactor" the code until it conforms to the simplicity criteria.
  - ➢ Repeat, "accumulating" unit tests over time.

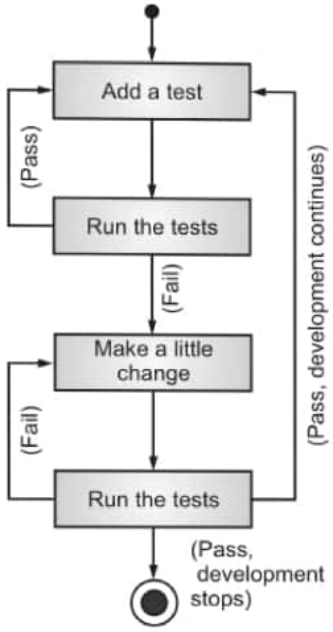


Fig. 5.9 : Basic TDD process

- The steps of test first development (TFD): First step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may first need to refactor any duplication out of your design as needed, turning TFD into TDD).
- TDD can be described with this simple formula:

  TDD = Refactoring + TFD.
- When you first go to implement a new feature, the first question that you ask is whether the existing design is the best design possible that enables you to implement that functionality. If so, you proceed via a TFD approach. If not, you refactor it locally to change the portion of the design affected by the new feature, enabling you to add that feature as easy as possible. As a result you will always be improving the quality of your design, thereby making it easier to work with in the future.

There are Two Levels of TDD:

1. **Acceptance TDD (ATDD):** With ATDD you write a single acceptance test, or behavioral specification depending on your preferred terminology, and then just enough production functionality/code to fulfill that test. The goal of ATDD is to specify detailed, executable requirements for your solution on a Just In Time (JIT) basis. ATDD is also called Behavior Driven Development (BDD).
2. **Developer TDD (DTDD):** With developer TDD you write a single developer test, sometimes inaccurately referred to as a unit test, and then just enough production code to fulfill that test. The goal of developer TDD is to specify a detailed, executable design for your solution on a JIT(Just in Time) basis. Developer TDD is often simply called TDD.

- TDD allows the process of writing tests and production code to steer the design. With TDD it is easy to imagine a clearer, cleaner method, class, or entire object model, refactor, protected the entire time by a solid suite of unit tests. As team learns about what actually works and what does not, they are in the best possible position to apply those insights.

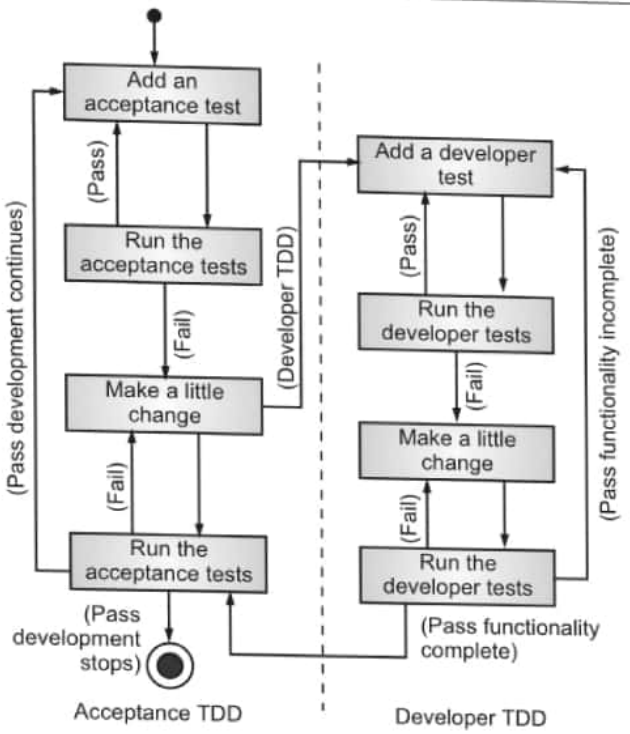


Fig. 5.10 : TDD process with ATDD and DTDD levels

Advantages

Early Bug Notification.

- Developers test their code but in the database world, this often consists of manual tests or one-off scripts. Using TDD you build up, over time, a suite of automated tests that you and any other developer can rerun at will.

Better Designed, Cleaner and More Extensible Code.

- It helps to understand how the code will be used and how it interacts with other modules.
- It results in better design decision and more maintainable code. TDD allows writing smaller code having single responsibility rather than monolithic procedures with multiple responsibilities. This makes the code simpler to understand.TDD also forces to write only production code to pass tests based on user requirements.

Confidence to Refactor

- If you refactor code, there can be possibilities of breaks in the code. So having a set of automated tests you can fix those breaks before release. Proper warning will be given if breaks found when automated tests are used. Using TDD, should results in faster, more extensible code with fewer bugs that can be updated with minimal risks.

Good for Teamwork

- In the absence of any team member, other team members can easily pick up and work on the code. It also aids knowledge sharing, thereby making the team more effective overall.

Good for Developers

- Though developers have to spend more time in writing TDD test cases, it takes a lot less time for debugging and developing new features. You will write cleaner, less complicated code.

5.4 RELEASE TESTING

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team. The primary goal of the release testing process is to convince the customer of the system that it is good enough for use. Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use. Release testing is usually a black-box testing process where tests are only derived from the system specification.
- Release testing is a form of system testing. Important differences:
  - ➢ A separate team that has not been involved in the system development, should be responsible for release testing.
  - ➢ System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).
- **Requirements-Based Testing** involves examining each requirement and developing a test or tests for it. It is validation rather than defect testing: you are trying to demonstrate that the system has properly implemented its requirements.
- **Scenario Testing** is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system. Scenarios should be realistic and real system users should be able to relate to them. If you have used scenarios as

part of the requirements engineering process, then you may be able to reuse these as testing scenarios.

- Part of release testing may involve testing the **emergent properties** of a system, such as performance and reliability. Tests should reflect the profile of use of the system. Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable. Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

5.4.1 Requirements-Based Testing

- Requirements-based testing is a testing approach in which test cases, conditions and data are derived from requirements. It includes functional tests and also non-functional attributes such as performance, reliability or usability.

Stages in Requirements Based Testing:

- **Defining Test Completion Criteria :** Testing is completed only when all the functional and non-functional testing is complete.
- **Design Test Cases :** A Test case has five parameters namely the initial state or precondition, data setup, the inputs, expected outcomes and actual outcomes.
- **Execute Tests :** Execute the test cases against the system under test and document the results.
- **Verify Test Results :** Verify if the expected and actual results match each other.
- **Verify Test Coverage :** Verify if the tests cover both functional and non-functional aspects of the requirement.
- **Track and Manage Defects :** Any defects detected during the testing process goes through the defect life cycle and are tracked to resolution. Defect Statistics are maintained which will give us the overall status of the project.

Requirements are Critical:

- Various studies have shown that software projects fail due to the following reasons:
  - ➢ Incomplete requirements and specifications
  - ➢ Frequent changes in requirements and specifications
  - ➢ When there is lack of user input to requirements
- Requirements based testing process addresses each of the above issues as follows :
  - ➢ The Requirements based testing process starts at the very early phase of the software development, as correcting issues/errors is easier at this phase.
  - ➢ It begins at the requirements phase as the chances of occurrence of bugs have its roots here.
  - ➢ It aims at quality improvement of requirements. Insufficient requirements leads to failed projects.

Example: Consider the Mentcare system requirements that are concerned with checking for drug allergies:

- If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.If a prescriber chooses to ignore an allergy warning, he or she shall provide a reason why this has been ignored.
- To check if these requirements have been satisfied, we need to develop several related tests:
  - ➢ Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
  - ➢ Set up a patient record with a known allergy. Prescribe the medication that the patient is allergic to and check that the warning is issued by the system.
  - ➢ Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
  - ➢ Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
  - ➢ Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

5.4.2 Scenario Testing

- **Scenario Testing** in software testing is a method in which actual scenarios are used for testing the software application instead of test cases. The purpose of scenario testing is to test end to end scenarios for a specific complex problem of the software. Scenarios

help in an easier way to test and evaluate end to end complicated problems.

- Test Scenarios are created for the following reasons:
  - ➢ Creating Test Scenarios ensures complete Test Coverage
  - ➢ Test Scenarios can be approved by various stakeholders like Business Analyst, Developers, Customers to ensure the Application Under Test is thoroughly tested. It ensures that the software is working for the most common use cases.
  - ➢ They serve as a quick tool to determine the testing work effort and accordingly create a proposal for the client or organize the workforce.
  - ➢ They help determine the most important end-to-end transactions or the real use of the software applications.
  - ➢ For studying the end-to-end functioning of the program, Test Scenario is critical.

Example 1 : Test Scenario for eCommerce Application

For an eCommerce Application, a few test scenarios would be

Test Scenario : Check the Login Functionality

Fig. 5.11

- In order to help you understand the difference Test Scenario and Test Cases, specific test cases for this Test Scenario would be
  - ➢ Check system behavior when valid email id and password is entered.
  - ➢ Check system behavior when *invalid* email id and *valid* password is entered.
  - ➢ Check system behavior when *valid* email id and *invalid* password is entered.
  - ➢ Check system behavior when *invalid* email id and *invalid* password is entered.
  - ➢ Check system behavior when email id and password are left blank and Sign in entered.
  - ➢ Check Forgot your password is working as expected
  - ➢ Check system behavior when valid/invalid phone number and password is entered.
  - ➢ Check system behavior when "Keep me signed" is checked

Example 2 : Scenario-Based Analysis of Wilderness Weather Station System:

- The weather station is composed of independent subsystem that communicates by broad casting messages on a conman infrastructure. Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.
- The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

Steps for Testing:

- Identify the objects, attributes and methods.
- Understand about the interactions between the system and its environment.

- Check the data processing and transmits ground thermometers, an anemometer, a wind vane a barometer and a rain gauge.
- Check the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client- server model.
- Check whether the command is issued to transmit the weather data, the weather station processes and summaries the collected data .the summation data is transmitted to the mapping computer when a request is received.

5.4.3 Performance Testing

- **Performances Testing** is a software testing process used for testing the speed, response time, stability, reliability, scalability and resource usage of a software application under particular workload. The main purpose of performance testing is to identify and eliminate the performance bottlenecks in the software application. It is a subset of performance engineering and also known as "Perf Testing".
- The focus of Performance Testing is checking a software program's
  - ➢ **Speed :** Determines whether the application responds quickly
  - ➢ **Scalability :** Determines maximum user load the software application can handle.
  - ➢ **Stability :** Determines if the application is stable under varying loads

Example Performance Test Cases

- Verify response time is not more than 4 secs when 1000 users access the website simultaneously.
- Verify response time of the Application Under Load is within an acceptable range when the network connectivity is slow
- Check the maximum number of users that the application can handle before it crashes.

5.5 USER TESTING

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing. This may involve formally testing a system that has been commissioned from an external supplier. It may be an informal process where users experiment with a new software product to see if they like it and to check that it does what they need.
- There are three different types of user testing:
  1. **Alpha Testing**: where a selected group of software users work closely with the development team to test early releases of the software.
  2. **Beta Testing:** where a release of the software is made available to a larger group of users to allow them to experiment and to raise problems that they discover with the system developers.
  3. **Acceptance Testing**: where customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

5.5.1 Alpha Testing

- **Alpha Testing** is a type of software testing performed to identify bugs before releasing the software product to the real users or public. It is a type of acceptance testing. The main objective of alpha testing is to refine the software product by finding and fixing the bugs that were not discovered through previous tests.
- This testing is referred to as an alpha testing only because it is done early on, near the end of the development of the software, and before Beta Testing. Alpha testing is typically performed by in-house software engineers or QA staff. It is the final testing stage before the software is released into the real world.
- Alpha testing has two phases,
  1. The first phase of testing is done by in-house developers. They either use hardware-assisted debuggers or debugger software. The aim to catch bugs quickly. Usually while alpha testing, a tester

will come across to plenty of bugs, crashes, missing features, and docs.

2. While the second phase of alpha testing is done by software QA staff, for additional testing in an environment. It involves both black box and White Box Testing.

Advantage of Alpha Testing

- Better insight about the software's reliability at its early stages
- Reduce delivery time to market
- Early feedback helps to improve software quality

5.5.2 Beta Testing

- This is a testing stage followed by the internal full alpha test cycle. This is the final testing phase where the companies release the software to a few external user groups outside the company test teams or employees. This initial software version is known as the beta version. Most companies gather user feedback in this release.
- **In Short, Beta Testing can be Defined as**– the testing carried out by real users in a real environment. Though companies do rigorous in-house quality assurance from dedicated test teams, it's practically impossible to test an application for each and every combination of the test environment. Beta releases make it easier to test the application on thousands of test machines and fix the issues before releasing the application to the public.
- The selection of beta test groups can be done based on the company's needs. The company can either invite few users to test the preview version of the application or they can release it openly to give it a try by any user. Fixing the issues in the beta release can significantly reduce the development cost as most of the minor glitches get fixed before the final release.

  Example: Recently Microsoft corporation released Windows 10 beta and based on the feedback from thousands of users they managed to release a stable OS version. In the past, Apple also released OS X beta in public and fixed many minor issues and improved the OS based on user feedback.

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

5.5.3 Acceptance Testing

- Formal testing with respect to users' needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customer or other authorized entity to determine whether or not to accept the system.

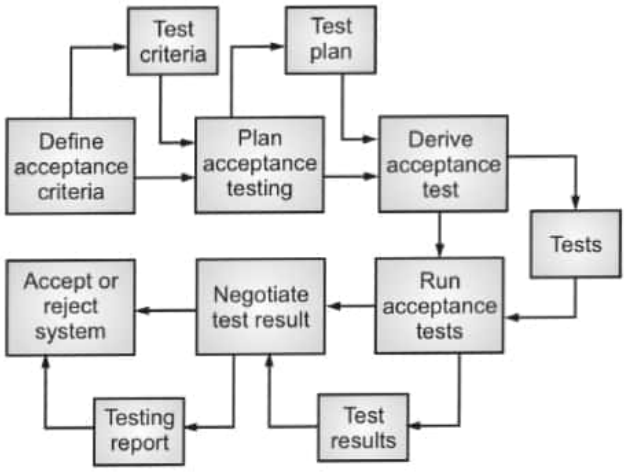


Fig. 5.12: Acceptance testing Process

- As shown in above fig. 5.12 there are six stages in the acceptance testing process:

1. **Define Acceptance Criteria :** This stage should ideally take place early in the process before the contract for the system is signed. The acceptance criteria should be part of the system contract and be approved by the customer and the developer.
2. **Plan Acceptance Testing:** This stage involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. The acceptance test plan should also discuss the required coverage of the requirements and the order in which system features are tested. It should define risks to the testing process such as system crashes and inadequate performance, and discuss how these risks can be mitigated.
3. **Derive Acceptance Tests :** Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system. They should ideally provide complete coverage of the system requirements.
4. **Run Acceptance Tests:** The agreed acceptance tests are executed on the system. Ideally, this step should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.
5. **Negotiate Test Results :** It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be used. They must also agree on how the developer will fix the identified problems.
6. **Reject/accept System :** Developers and the customer meets to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems and repeat the acceptance testing phase.

EXERCISE

1. What is software testing? Why it is important?
2. What is the difference between validation and verification?
3. Bring out the differences between testing and inspection.
4. Explain software inspection with its advantages.
5. Explain with neat diagram model of the software testing process
6. Explain the different levels in Development testing.
7. Write short note on

   a)Unit testing

   b)component testing

   c)System Testing
8. Explain unit testing with example.
9. Explain how to choose unit test cases?
10. Explain interface testing with suitable diagram.
11. Explain System testing with example.
12. Explain Test driven development with neat diagram. Also mention its advantages.

13. What is the release testing? What are the differences between release testing and system testing?
14. What are the benefits of involving users in release testing at an early stage in the testing process? Are there disadvantages in user involvement?
15. Explain with example Requirements-based testing. Mention the issues resolved during it.
16. Write a scenario that could be used to help design tests for the wilderness weather station system.
17. What is performance testing? Give examples of performance test cases.
18. Explain different types of user testing.
19. What is acceptance testing? Why it is important?
20. Compare alpha - beta testing?
21. With the help of neat diagram explain six stages of Acceptance testing process.

✠ ✠ ✠

UNIT VI

SYSTEM DEPENDABILITY AND SECURITY

6.1 DEPENDABILITY PROPERTIES

Introduction:

- As computer systems have become deeply embedded in our business and personal lives, the problems that result from system and software failure are increasing. A failure of server software in an e-commerce company could lead to a major loss of revenue and customers for that company. A software error in an embedded control system in a car could lead to expensive recalls of that model for repair and, in the worst case, could be a contributory factor in accidents. The infection of company PCs with malware requires expensive clean-up operations to sort out the problem and could lead to the loss of or damage to sensitive information.
- Because software-intensive systems are so important to governments, companies, and individuals, we have to be able to trust these systems. The software should be available when it is needed, and it should operate correctly without undesirable side effects, such as unauthorized information disclosure. In short, we should be able to depend on our software systems. The term dependability was proposed by Jean-Claude Laprie in 1995 to cover the related systems attributes of availability, reliability, safety, and security. His ideas were revised over the next few years and are discussed in a definitive paper published in 2004 (Avizienis et al. 2004).

Definition:

- "In **software engineering**, **dependability** is the ability to provide services that can defensibly be trusted within a time-period. This may also encompass mechanisms designed to increase and maintain the **dependability** of a system or **software**."
- The dependability of systems is usually more important than their detailed functionality for the following reasons:
  - ➢ If System will get failures it affect a large number of people and functionality. Many systems include functionality that is rarely used. If this functionality were left out of the system, only a small number of users would be affected. System failures that affect the availability of a system potentially affect all users of the system. Unavailable systems may mean that normal business is impossible.
  - ➢ Users often reject systems that are unreliable, unsafe, or insecure If users find that a system is unreliable or insecure, they will refuse to use it. Furthermore, they may also refuse to buy or use other products from the company that produced the unreliable system. They do not want a repetition of their bad experience with an undependable system.
  - ➢ System failure costs may be enormous For some applications, such as a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system. Failures in systems that control critical infrastructure such as the power network have widespread economic consequences.
  - ➢ Undependable systems may cause information loss Data is very expensive to collect and maintain; it is usually worth much more than the computer system on which it is processed. The cost of recovering lost or corrupt data is usually very high.
- Building dependable software is part of the more general process of dependable systems engineering.. It executes in an operational environment that includes the hardware on which the software executes, the

human users of that software and the organizational or business processes where the software is used.

When designing a dependable system, you therefore have to consider:

- Hardware failure System hardware may fail because of mistakes in its design, because components fail as a result of manufacturing errors, because of environmental factors such as dampness or high temperatures, or because components have reached the end of their natural life.
- Software failure System software may fail because of mistakes in its specification, design, or implementation.
- Operational failure Human users may fail to use or operate the system as intended by its designers. As hardware and software have become more reliable, failures in operation are now, perhaps, the largest single cause of system failures.
- These failures are often interrelated. A failed hardware component may mean system operators have to cope with an unexpected situation and additional workload. This puts them under stress, and people under stress often make mistakes. These mistakes can cause the software to fail, which means more work for operators, even more stress, and so on. As a result, it is particularly important that designers of dependable, software intensive systems take a holistic socio technical systems perspective rather than focus on a single aspect of the system such as its software or hardware
- The dependability of a computer system is a property of the system that reflects its trustworthiness. Trustworthiness here essentially means the degree of confidence a user has that the system will operate as they expect and that the system will not "fail" in normal use. It is not meaningful to express dependability numerically.
- Rather, relative terms such as "not dependable," "very dependable," and "ultra dependable" can reflect the degree of trust that we might have in a system. There are five principal dimensions to dependability,

1. **Availability :** the availability of a system is the probability that it will be up and running and able to deliver useful services to users at any given time.
2. **Reliability** : Informally, the reliability of a system is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.
3. **Safety** : Informally, the safety of a system is a judgment of how likely it is that the system will cause damage to people or its environment.
4. **Security** Informally, the security of a system is a judgment of how likely it is that the system can resist accidental or deliberate intrusions.
5. **Resilience** Informally, the resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyber attacks.

- The dependability properties shown in Fig. 6.1 are complex properties that can be broken down into several simpler properties. For example, security includes "integrity" (ensuring that the systems program and data are not damaged) and "confidentiality" (ensuring that information can only be accessed by people who are authorized). Reliability includes "correctness" (ensuring the system services are as specified), "precision" (ensuring information is delivered at an appropriate level of detail), and "timeliness" (ensuring that information is delivered when it is required). Of course, not all dependability properties are critical for all systems.

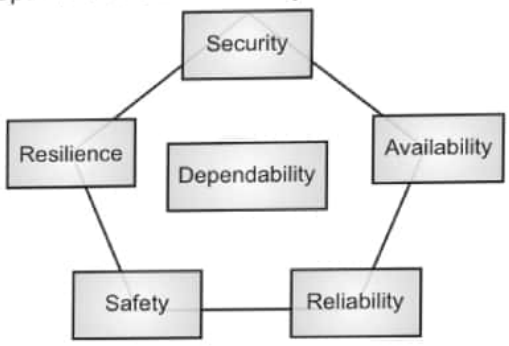


Fig. 6.1 : System Dependability Key Pillars

6.2 AVAILABILITY

- **Availability** is a probability measure. It outlines the likelihood that the system will be operational at any given point in time in order to fulfill requests. You can't trust a system that is frequently down, or the data you seek is consistently unavailable.
- When measuring availability, you must take into account the time it takes to repair failures, and the average time between repairs. The resultant equation is *uptime / (uptime + downtime)*. Availability does not account for preventative maintenance, however, because this is built into the life cycle of the system.

Resilience:

- Resilience Informally, the resilience of a system is a judgment of how well that system can maintain the continuity of its critical services in the presence of disruptive events, such as equipment failure and cyber attacks. Resilience is a more recent addition to the set of dependability properties that were originally suggested by Laprie.
- Other system properties are closely related to these five dependability properties and influence a system's dependability:

1. **Repairability :** System failures are inevitable, but the disruption caused by failure can be minimized if the system can be repaired quickly. It must be possible to diagnose the problem, access the component that has failed, and make changes to fix that component. Repair ability in software is enhanced when the organization using the system has access to the source code and has the skills to make changes to it. Open-source software makes this easier, but the reuse of components can make it more difficult.
2. **Maintainability :** As systems are used, new requirements emerge, and it is important to maintain the value of a system by changing it to include these new requirements. Maintainable software is software that can be adapted economically to cope with new requirements, and where there is a low probability that making changes will introduce new errors into the system.
3. **Error Tolerance :** This property can be considered as part of usability and reflects the extent to which the system has been designed, so that user input errors are avoided and tolerated. When user errors occur, the system should, as far as possible, detect these errors and either fix them automatically or request the user to re-input their data.

- To develop dependable software, you therefore need to ensure that:
  - ➢ You avoid the introduction of accidental errors into the system during software specification and development.
  - ➢ You design verification and validation processes that are effective in discovering residual errors that affect the dependability of the system.
  - ➢ You design the system to be fault tolerant so that it can continue working when things go wrong.
  - ➢ You design protection mechanisms that guard against external attacks that can compromise the availability or security of the system.
  - ➢ You configure the deployed system and its supporting software correctly for its operating environment.
  - ➢ You include system capabilities to recognize external cyber attacks and to resist these attacks.
  - ➢ You design systems so that they can quickly recover from system failures and cyber attacks without the loss of critical data.
- Fig. 6.2 shows the relationship between costs and incremental improvements in dependability. If your software is not very dependable, you can get significant improvements fairly cheaply by using better software engineering. However, if you are already using good practice, the costs of improvement are much greater, and the benefits from that improvement are less.
  - ➢ Sociotechnical systems 291 fewer failures. Consequently, more and more tests are needed to try and assess how many problems remain in the software. Testing is a very expensive process, so this can significantly increase the cost of high-dependability systems.

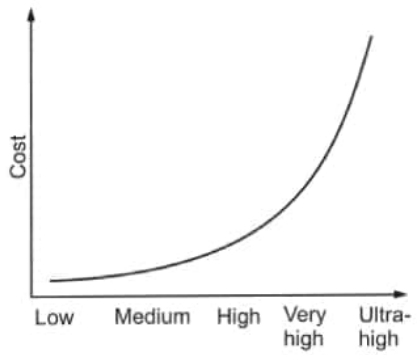


Fig. 6.2 : Cost dependability Curve

6.3 RELIABILITY

- The objective of this chapter is to explain how software reliability may be specified, implemented, and measured.
- You may think that availability and reliability are the same, but they differ. **Reliability** is a quantitative measurement (as is availability), but it outlines the probability that the system will run without failure over a given time. This does not mean that the right data is served, only the probability that the system will continue to run.
- Technically, you could have a lemon of a system that is still available. You may need to add a quick fix or patch to get it up and running, but it still contains the data.
- When you will read this you will be able to :
  - ➢ understand the distinction between software reliability and software availability;
  - ➢ have been introduced to metrics for reliability specification and how these are used to specify measurable reliability requirements;
  - ➢ understand how different architectural styles may be used to implement reliable, fault-tolerant systems architectures;
  - ➢ know about good programming practice for reliable software engineering;
  - ➢ understand how the reliability of a software system may be measured using statistical testing.
- Our dependence on software systems for almost all aspects of our business and personal lives means that we expect that software to be available when we need it. This may be early in the morning or late at night, at weekends or during holidays the software must run all day, every day of the year.
- We expect that software will operate without crashes and failures and will preserve our data and personal information. We need to be able to trust the software that we use, which means that the software must be reliable. The use of software engineering techniques, better programming languages, and effective quality management has led to significant improvements in software reliability over the past 20 years. Nevertheless, system failures still occur that affect the system's availability or lead to incorrect results being produced. In situations where software has a particularly critical role perhaps in an aircraft or as part of the national critical infrastructure special reliability engineering techniques may be used to achieve the high levels of reliability and availability that are required.
- Brian Randell, a pioneer researcher in software reliability, defined a fault–error–failure model (Randell 2000) based on the notion that human errors cause faults; faults lead to errors, and errors lead to system failures.
- He defined these terms precisely:
  - ➢ **Human Error or Mistake:** Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
  - ➢ **System Fault :** A characteristic of a software system that can lead to a system error. The fault in the above example is the inclusion of code to add 1 to a variable called Transmission time, without a check to see if the value of Transmission time is greater than or equal to 23.00.
  - ➢ **System Error :** An erroneous system state during execution that can lead to system behaviour that is unexpected by system users. In this example, the value of the variable Transmission time is set

incorrectly to 24.XX rather than 00.XX when the faulty code is executed.

➢ **System Failure :** An event that occurs at some point in time when the system does not deliver a service as expected by its users. In this case, no weather data is transmitted because the time is invalid.

- The distinction between faults, errors, and failures leads to three complementary approaches that are used to improve the reliability of a system:

➢ **Fault Avoidance :** The software design and implementation process should use approaches to software development that help avoid design and programming errors and so minimize the number of faults introduced into the system. Fewer faults means less chance of runtime failures. Fault-avoidance techniques include the use of strongly typed programming language to allow extensive compiler checking and minimizing the use of error-prone programming language constructs, such as pointers.

➢ **Fault Detection and Correction :** Verification and validation processes are designed to discover and remove faults in a program, before it is deployed for operational use. Critical systems require extensive verification and validation to discover as many faults as possible before deployment and to convince the system stakeholders and regulators that the system is dependable. Systematic testing and debugging and static analysis are examples of fault-detection techniques.

➢ **Fault Tolerance :** The system is designed so that faults or unexpected system behavior during execution are detected at runtime and are managed in such a way that system failure does not occur. Simple approaches to fault tolerance based on built-in runtime checking may be included in all systems..

- Unfortunately, applying fault-avoidance, fault-detection, and fault-tolerance techniques is not always cost-effective. The cost of finding and removing the remaining faults in a software system rises exponentially as program faults are discovered and removed .

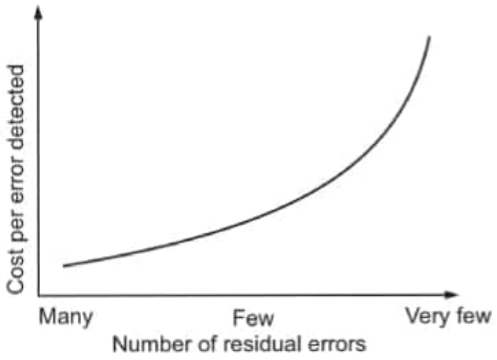


Fig. 6.3 : The increasing costs of residual fault removal

- As a result, software companies accept that their software will always contain some residual faults. The level of faults depends on the type of system. Software products have a relatively high level of faults, whereas critical systems usually have a much lower fault density. The rationale for accepting faults is that, if and when the system fails, it is cheaper to pay for the consequences of failure than it would be to discover and remove the faults before system delivery. However, the decision to release faulty software is not simply an economic one. The social and political acceptability of system failure must also be taken into account.
- More precise definitions of availability and reliability are:

➢ **Reliability :** The probability of failure-free operation over a specified time, in a given environment, for a specific purpose.

➢ **Availability :** The probability that a system, at a point in time, will be operational and able to deliver the requested services.

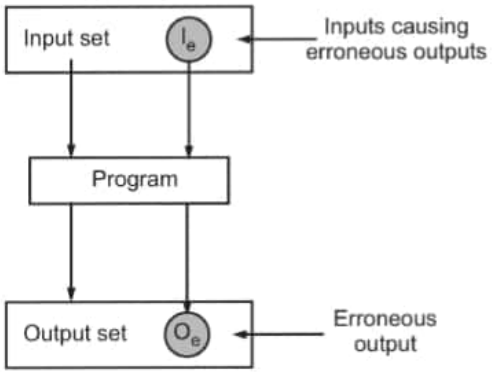


Fig. 6.4 : A system as an input/output mapping

Explanation:

- System reliability is not an absolute value it depends on where and how that system is used. For example, let's say that you measure the reliability of an application in an office environment where most users are uninterested in the operation of the software. They follow the instructions for its use and do not try to experiment with the system. If you then measure the reliability of the same system in a university environment, then the reliability may be quite different. Here, students may explore the boundaries of the system and use it in unexpected ways. This may result in system failures that did not occur in the more constrained office environment. Therefore, the perceptions of the system's reliability in each of these environments are different. The above definition of reliability is based on the idea of failure-free operation, where failures are external events that affect the users of a system. But what constitutes "failure"? A technical definition of failure is behavior that does not conform to the system's specification.
- However, there are two problems with this definition:
  1. Software specifications are often incomplete or incorrect, and it is left to software engineers to interpret how the system should behave. As they are not domain experts, they may not implement the behavior that users expect. The software may behave as specified, but, for users, it is still failing.
  2. No one except system developers reads software specification documents. Users may therefore anticipate that the software should behave in one way when the specification says something completely different.
- This incident shows that system dependability does not just depend on good engineering. It also requires attention to detail when the system requirements are derived and the specification of software requirements that are geared to ensuring the dependability of a system.
- Those dependability requirements are of two types:
  1. Functional requirements, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks.
  2. Non-functional requirements, which define the required reliability and availability of the system

Table 6.1 : Availability Specification

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1440 minutes), the system will be unavailable for 144 minutes.
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period-roughly, one minute per week.

- Three metrics may be used to specify reliability and availability:
  1. Probability of failure on demand (POFOD) If you use this metric, you define the probability that a demand for service from a system will result in a system failure. So, POFOD = 0.001 means that there is a 1/1000 chance that a failure will occur when a demand is made.
  2. Rate of occurrence of failures (ROCOF) This metric sets out the probable number of system failures that are likely to be observed relative to a certain time period (e.g., an hour), or to the number of system executions. In the example above, the ROCOF is 1/1000. The reciprocal of ROCOF is the mean time to failure (MTTF), which is sometimes used as a reliability metric. MTTF is the average number of time units between observed system failures. A ROCOF of two failures per hour implies that the mean time to failure is 30 minutes.
  3. Availability (AVAIL) AVAIL is the probability that a system will be operational when a demand is made for service. Therefore, an availability of 0.9999 means that, on average, the system will be available for 99.99% of the operating time. Figure 11.4 shows what different levels of availability mean in practice

Non-Functional Reliability Requirements

Quantitative Reliability Specification is Useful in a Number of Ways:

- The process of deciding the required level of the reliability helps to clarify what stakeholders really need. It helps stakeholders understand that there are different types of system failure, and it makes clear to them that high levels of reliability are expensive to achieve.
- It provides a basis for assessing when to stop testing a system. You stop when the system has reached its required reliability level.
- It is a means of assessing different design strategies intended to improve the reliability of a system. You can make a judgment about how each strategy might lead to the required levels of reliability.
- If a regulator has to approve a system before it goes into service (e.g., all systems that are critical to flight safety on an aircraft are regulated), then evidence that a required reliability target has been met is important for system certification.

Reliability Measurement

- To assess the reliability of a system, you have to collect data about its operation. The data required may include:
  - ➢ The number of system failures given a number of requests for system services. This is used to measure the POFOD and applies irrespective of the time over which the demands are made.

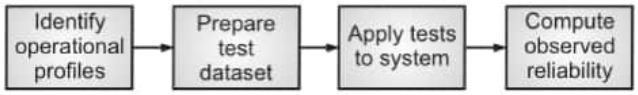


Fig. 6.5 : Statistical testing for reliability measurement

  - ➢ The time or the number of transactions between system failures plus the total elapsed time or total number of transactions. This is used to measure ROCOF and MTTF.
  - ➢ The repair or restart time after a system failure that leads to loss of service. This is used in the measurement of availability. Availability does not just depend on the time between failures but also on the time required to get the system back into operation.

6.4 SAFETY

- The objective of this chapter is to explain techniques that are used to ensure safety when developing critical systems.
- Today, almost everything has some sort of computer or software component. Examples include smart phones, thermostats, fridges and medical devices. **Safety** is a property that indicates a system's ability to run without posing harm to humans or to the system itself.
- As opposed to stated requirements (e.g. what a system WILL do), safety specifications state what the system will NOT do. For example, there will not be a single point of failure, the system will not a dminister medication if any input is corrupt, etc.
- When you have read this chapter, you will able to learn:
  - ➢ understand what is meant by a safety-critical system and why safety has to be considered separately from reliability in critical systems engineering;
  - ➢ understand how an analysis of hazards can be used to derive safety requirements;
  - ➢ know about processes and tools that are used for software safety assurance;
  - ➢ understand the notion of a safety case that is used to justify the safety of a system to regulators, and how formal arguments may be used in safety cases.
- A system can be considered to be safe if it operates without catastrophic failure, that is, failure that causes or may cause death or injury to people. Systems whose failure may lead to environmental damage may also be safety-critical as environmental damage (such as a chemical leak) can lead to subsequent human injury or death.
- Software in safety-critical systems has a dual role to play in achieving safety:
  - ➢ The system may be software-controlled so that the decisions made by the software and subsequent actions are safety-critical. Therefore, the software

behavior is directly related to the overall safety of the system.

- ➢ Software is extensively used for checking and monitoring other safety-critical components in a system. For example, all aircraft engine components are monitored by software looking for early indications of component failure. This software is safety critical because, if it fails, other components may fail and cause an accident. Safety in software systems is achieved by developing an understanding of the situations that might lead to safety-related failures.

- The software is engineered so that such failures do not occur. You might therefore think that if a safety-critical system is reliable and behaves as specified, it will therefore be safe. Unfortunately, it isn't quite as simple as that. System reliability is necessary for safety achievement, but it isn't enough. Reliable systems can be unsafe and vice versa.
- The Warsaw Airport accident was an example of such a situation.
- Software systems that are reliable may not be safe for four reasons:
  - ➢ We can never be 100% certain that a software system is fault-free and fault-tolerant. Undetected faults can be dormant for a long time, and software failures can occur after many years of reliable operation.
  - ➢ The specification may be incomplete in that it does not describe the required behavior of the system in some critical situations.
  - ➢ Hardware malfunctions may cause sensors and actuators to behave in an unpredictable way. When components are close to physical failure, they may behave erratically and generate signals that are outside the ranges that can be handled by the software. The software may then either fail or wrongly interpret these signals.
  - ➢ The system operators may generate inputs that are not individually incorrect but that, in some situations, can lead to a system malfunction. An anecdotal example of this occurred when an aircraft undercarriage collapsed while the aircraft was on the ground.
- Apparently, a technician pressed a button that instructed the utility management software to raise the undercarriage. The software carried out the mechanic's instruction perfectly. However, the system should have disallowed the command unless the plane was in the air

Safety-Critical Systems:

- Safety-critical systems are systems in which it is essential that system operation is always safe. That is, the system should never damage people or the system's environment, irrespective of whether or not the system conforms to its specification. Examples of safety-critical systems include control and monitoring systems in aircraft, process control systems in chemical and pharmaceutical plants, and automobile control systems.
- Safety-critical software falls into two classes:
  1. Primary safety-critical software This is software that is embedded as a controller in a system. Malfunctioning of such software can cause a hardware malfunction, which results in human injury or environmental damage.. The insulin pump system is a simple system, but software control is also used in very complex safety-critical systems. Software rather than hardware control is essential because of the need to manage large numbers of sensors and actuators, which have complex control laws.

     For example, advanced, aerodynamically unstable, military aircraft require continual software-controlled adjustment of their flight surfaces to ensure that they do not crash.
  2. Secondary safety-critical software This is software that can indirectly result in an injury.
- Safety engineering whose malfunctioning might result in a design fault in the object being designed. This fault may cause injury to people if the designed system malfunctions. Another example of a secondary safety-

critical system is the Mentcare system for mental health patient management. Failure of this system, whereby an unstable patient may not be treated properly, could lead to that patient injuring himself or

Table 6.2 : Safety Terminology

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events that results in human death or injury, damage to property or to the environment. An overdose of insulin is an example of an accident.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could lead to serious injury or the death of the user of the insulin pump.
Hazard	A condition with the potential for causing or contributing to an accident A failure of the sensor that measures blood glucose is an example of a hazard.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from -probable- (say 1/100 chance of a hazard occurring) to 'implausible" (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that overestimates the user's blood sugar level is low.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are kilted, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is "very high."
Risk	A measure of the probability that the system will cause an accident The risk is assessed by considering the hazard probability, the hazard severity., and the probability that the hazard will lead to an accident. The risk of an insulin overdose is medium to low.

Safety Requirements:

- Safety requirements are functional requirements, which define checking and recovery facilities that should be included in the system and features that provide protection against system failures and external attacks. The starting point for generating functional safety requirements is usually domain knowledge, safety standards, and regulations. These lead to high-level requirements that are perhaps best described as "shall not" requirements. By contrast with normal functional requirements that define what the system shall do, "shall not" requirements define system behavior that is unacceptable.
- Examples of "shall not" requirements are: "The system shall not allow reverse thrust mode to be selected when the aircraft is in flight." "The system shall not allow the simultaneous activation of more than three alarm signals." "The navigation system shall not allow users to set the required destination when the car is moving." These "shall not" requirements cannot be implemented directly but have to be decomposed into more specific software functional requirements. Alternatively, they may be implemented through system design decisions such as a decision to use particular types of equipment in the system.

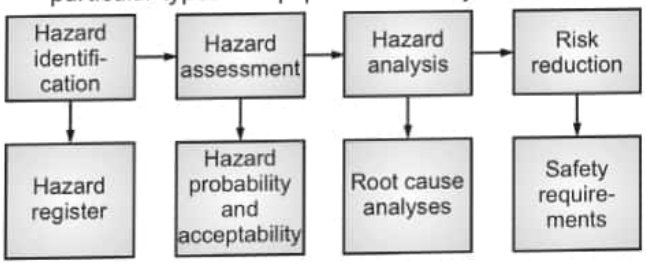


Fig. 6.6 : Hazard driven requirements specification

- There are four activities in a hazard-driven safety specification process:
  1. Hazard identification The hazard identification process identifies hazards that may threaten the system. These hazards may be recorded in a hazard register. This is a formal document that records the safety analyses and assessments and that may be submitted to a regulator as part of a safety case.

2. Hazard assessment The hazard assessment process decides which hazards are the most dangerous and/or the most likely to occur. These should be prioritized when deriving safety requirements.
3. Hazard analysis This is a process of root-cause analysis that identifies the events that can lead to the occurrence of a hazard.
4. Risk reduction This process is based on the outcome of hazard analysis and leads to identification of safety requirements.

- These requirements may be concerned with ensuring that a hazard does not arise or lead to an accident or that if an accident does occur, the associated damage is minimized.
  - ➢ Three conditions could lead to the administration of an incorrect dose of insulin.
    1. The level of blood sugar may have been incorrectly measured, so the insulin requirement has been computed with an incorrect input.

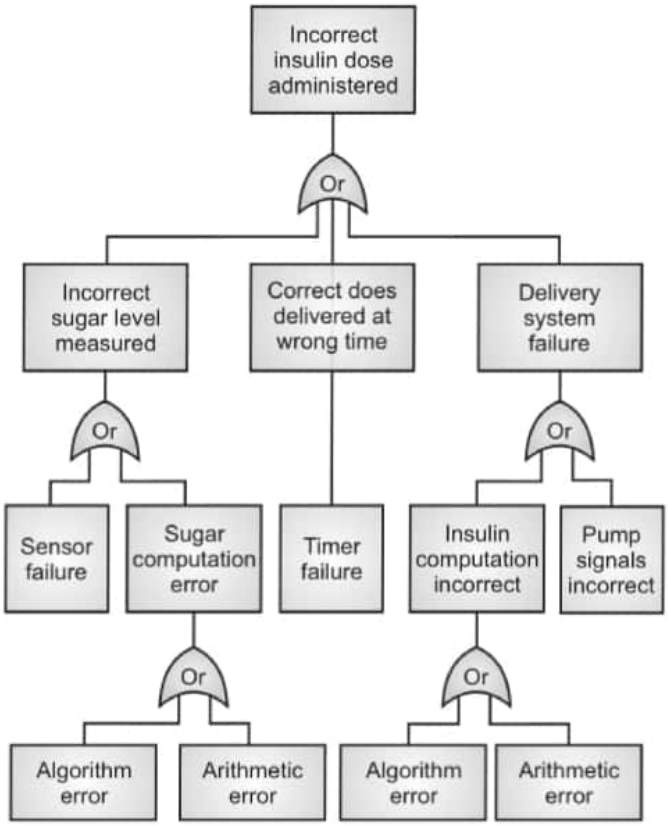


Fig. 6.7 : An example of a fault tree

2. The delivery system Alternatively,
3. The dose may be correctly computed, but it is delivered too early or too late.

  - ➢ The left branch of the fault tree, concerned with incorrect measurement of the blood sugar level, identifies how this might happen. This could occur either because the sensor that provides an input to calculate the sugar level has failed or because the calculation of the blood sugar level has been carried out incorrectly. The sugar level is calculated from some measured parameter, such as the conductivity of the skin. Incorrect computation can result from either an incorrect algorithm or an arithmetic error that results from the use of floating-point numbers.
  - ➢ The central branch of the tree is concerned with timing problems and concludes that these can only result from system timer failure. Incorrect sugar level measured Incorrect insulin dose administered or Correct dose delivered at wrong time Sensor failure or Sugar computation error Timer failure Pump signals incorrect or Insulin computation incorrect
  - ➢ The right branch of the tree, concerned with delivery system failure, examines possible causes of this failure. These could result from an incorrect computation of the insulin requirement or from a failure to send the correct signals to the pump that delivers the insulin. Again, an incorrect computation can result from algorithm failure or arithmetic errors.

Safety Cases

- This evidence is collected during the systems development process. It may include information about hazard analysis and mitigation, test results, static analyses, information about the development processes used, records of review meetings, and so on. It is assembled and organized into a safety case, a detailed presentation of why the system owners and developers believe that a system is safe.

- A safety case is a set of documents that includes a description of the system to be certified, information about the processes used to develop the system, and, critically, logical arguments that demonstrate that the system is likely to be safe. More succinctly, Bishop and Bloomfield (Bishop and Bloomfield 1998) define a safety case as: A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment†. The organization and contents of a safety case depend on the type of system that is to be certified and its context of operation Safety case structures vary, depending on the industry and the maturity of the domain. For example, nuclear safety cases have been required for many years.
- A safety case refers to a system as a whole, and, as part of that case, there may be a subsidiary software safety case. When constructing a software safety case, you have to relate software failures to wider system failures and demonstrate either that these software failures will not occur or that they will not be propagated in such a way that dangerous system failures may occur.

Table 6.3: Safety Case

Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements taken from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk. Hazard analyses and hazard logs.
Design analysis	A set of structured arguments that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and,, where appropriate, the test plans for the system. Summaries of the test results showing defects that have been detected and corrected.
	If formal methods have been used, a formal system specification and any analyses of that specification. Records of static analyses of the source code.
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes carried out during system development.
Change management processes	Records of all changes proposed, actions taken, and, where appropriate, justification of the safety of these changes. Information about configuration procedures and configuration management logs.
Associated safety cases	References to other safety cases that may impact the safety case.

- Safety cases are large and complex documents, and so they are very expensive to produce and maintain. Because of these high costs, safety-critical system developers have to take the requirements of the safety case into account in the development process: 1. Graydon et al. (Graydon, Knight, and Strunk 2007) argue that the development of a safety case should be tightly integrated with system design and implementation. This means that system design decisions may be influenced by the requirements of the safety case.

6.5 SECURITY

- Given prominent security breaches in the news, we've placed security at the top of the list. **Security** is a property that denotes the ability for a system to shield itself from attack. Today's systems are connected to the Internet, which means the threat from external attack is far greater. The highest risk to a given system, however, is its users. Through social engineering, weak passwords, and inadvertent or deliberate sharing of system information, employees can be the breech in the dam.

- When it comes to the security of a system, think of the acronym CIA:
  - **C - Confidentiality:** Access is only granted to authorized users.
  - **I - Integrity:** You can't alter data if you are not the owner.
  - **A - Availability:** The data in the system is consistently accessible.
- Security is inexorably linked to the next three facets of dependability. If a system is not secure, then availability, reliability and safety measures go right out the window.
- When you have read this chapter, you will:
  - understand the importance of security engineering and the difference between application security and infrastructure security;
  - know how a risk-based approach can be used to derive security requirements and analyze system designs;
  - know of software architectural patterns and design guidelines for secure systems engineering;
  - understand why security testing and assurance is difficult and expensive.
- You have to take three security dimensions into account in secure systems engineering:
  - Confidentiality Information in a system may be disclosed or made accessible to people or programs that are not authorized to have access to that information. For example, the theft of credit card data from an e-commerce system is a confidentiality problem.
  - Integrity Information in a system may be damaged or corrupted, making it unusual or unreliable. For example, a worm that deletes data in a system is an integrity problem.
  - Availability Access to a system or its data that is normally available may not be possible.
- A denial-of-service attack that overloads a server is an example of a situation where the system availability is compromised. These dimensions are closely related. If an attack makes the system unavailable, then you will not be able to update information that changes with time. This means that the integrity of the system may be compromised. If an attack succeeds and the integrity of the system is compromised, then it may have to be taken down to repair the problem. Therefore, the availability of the system is reduced.
- From an organizational perspective, security has to be considered at three levels:
  1. Infrastructure security, which is concerned with maintaining the security of all systems and networks that provide an infrastructure and a set of shared services to the organization.
  2. Application security, which is concerned with the security of individual application systems or related groups of systems.
  3. Operational security, which is concerned with the secure operation and use of the organization's systems

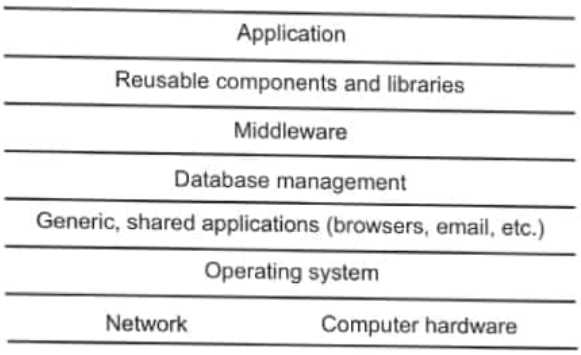


Fig. 6.8 : System layers where security may be compromised

  - an operating system platform, such as Linux or Windows;
  - other generic applications that run on that system, such as web browsers and email clients;
  - a database management system;
  - middleware that supports distributed computing and database access; and
  - libraries of reusable components that are used by the application software
- System vulnerabilities may arise because of requirements, design, or implementation problems, or they may stem from human, social, or organizational

failings. People may choose easy-to-guess passwords or write down their passwords in places where they can be found. System administrators make errors in setting up access control or configuration files, and users don't install or use protection software. However, we cannot simply class these problems as human errors. User mistakes or omissions often reflect poor systems design decisions that require, for example, frequent password changes (so that users write down their passwords) or complex configuration mechanisms.

Table 6.4 : Examples of Security Terminology

Term	Example
Asset	The record of each patient who is receiving or has received treatment.
Attack	An impersonation of an authorized user.
Control	A password checking system that disallows user passwords that are proper names or words that are normally included in a dictionary.
Exposure	Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data. Financial loss from legal action by the sports star. Loss of reputation.
Treat	An unauthorized user will access to the system by guessing the credentials (login name and password) of an authorized user.
Vulnerability	Authentication is based on a password system that does not require strong passwords. Users can then set easily guessable passwords.

- Four types of security threats may arise:
  1. Interception threats that allow an attacker to gain access to an asset. So, a possible threat to the Mentcare system might be a situation where an attacker gains access to the records of an individual patient.
  2. Interruption threats that allow an attacker to make part of the system unavailable. Therefore, a possible threat might be a denial-of-service attack on a system database server.
  3. Modification threats that allow an attacker to tamper with a system asset. In the Mentcare system, a modification threat would be where an attacker alters or destroys a patient record.
  4. Fabrication threats that allow an attacker to insert false information into a system.
- This is perhaps not a credible threat in the Mentcare system but would certainly be a threat in a banking system, where false transactions might be added to the system that transfers money to the perpetrator's bank account.
- The controls that you might put in place to enhance system security are based on the fundamental notions of avoidance, detection, and recovery:
  - ➢ **Vulnerability Avoidance :** Controls that are intended to ensure that attacks are unsuccessful. The strategy here is to design the system so that security problems are avoided. For example, sensitive military systems are not connected to the Internet so that external access is more difficult. You should also think of encryption as a control based on avoidance. Any unauthorized access to encrypted data means that the attacker cannot read the encrypted data. It is expensive and time consuming to crack strong encryption.
  - ➢ **Attack Detection and Neutralization:** Controls that are intended to detect and repel attacks. These controls involve including functionality in a system that monitors its operation and checks for unusual patterns of activity.
  - ➢ **Exposure Limitation and Recovery:** Controls that support recovery from problems. These can range from automated backup strategies and information "mirroring" through to insurance policies that cover the costs associated with a successful attack on the system.
- Security is closely related to the other dependability attributes of reliability, availability, safety, and resilience:
  - ➢ **Security and Reliability:** If a system is attacked and the system or its data are corrupted as a

consequence of that attack, then this may induce system failures that compromise the reliability of the system. Errors in the development of a system can lead to security loopholes. If a system does not reject unexpected inputs or if array bounds are not checked, then attackers can exploit these weaknesses to gain access to the system. For example, failure to check the validity of an input may mean that an attacker can inject and execute malicious code.

- **Security and Availability:** A common attack on a web-based system is a denial of-service attack, where a web server is flooded with service requests from a range of different sources. The aim of this attack is to make the system unavailable. A variant of this attack is where a profitable site is threatened with this type of attack unless a ransom is paid to the attackers.
- **Security and Safety:** Again, the key problem is an attack that corrupts the system or its data. Safety checks are based on the assumption that we can analyze the source code of safety-critical software and that the executing code is a completely accurate translation of that source code. If this is not the case, because an attacker has changed the executing code, safety-related failures may be induced and the safety case made for the software is invalid. Like safety, we cannot assign a numeric value to the security of a system, nor can we exhaustively test the system for security. Both safety and security can be thought of as "negative" or "shall not" characteristics in that they are concerned with things that should not happen. As we can never prove a negative, we can never prove that a system is safe or secure.
- **Security and Resilience:** Resilience, covered in system characteristic that reflects its ability to resist and recover from damaging events. The most probable damaging event on networked software systems is a cyber attack of some kind, so most of the work now done in resilience is aimed at deterring, detecting, and recovering from such attacks.

Security and Organizations:

- Security risk management is therefore a business rather than a technical issue. It has to take into account the financial and reputational losses from a successful system attack as well as the costs of security procedures and technologies that may reduce these losses.
- For risk management to be effective, organizations should have a documented information security policy that sets out:
  - The assets that must be protected It does not necessarily make sense to apply stringent security procedures to all organizational assets. Many assets are not confidential, and a company can improve its image by making these assets freely available. The costs of maintaining the security of information that is in the public domain are much less than the costs of keeping confidential information secure.
  - The level of protection that is required for different types of assets Not all assets need the same level of protection. In some cases (e.g., for sensitive personal information), a high level of security is required; for other information, the consequences of loss may be minor, so a lower level of security is adequate. Therefore, some information may be made available to any authorized and logged-in user; other information may be much more sensitive and only available to users in certain roles or positions of responsibility.
  - The responsibilities of individual users, managers, and the organization The security policy should set out what is expected of users for example, use strong passwords, log out of computers, and lock offices. It also defines what users can expect from the company, such as backup and information-archiving services, and equipment provision.
  - Existing security procedures and technologies that should be maintained For reasons of practicality and cost, it may be essential to continue to use

existing approaches to security even where these have known limitations. For example, a company may require the use of a login name/password for authentication, simply because other approaches are likely to be rejected by users.

- Risk assessment and management is an organizational activity rather than a technical activity that is part of the software development life cycle. The reason for this is that some types of attack are not technology-based but rather rely on weaknesses in more general organizational security. For example, an attacker may gain access to equipment by pretending to be an accredited engineer. If an organization has a process to check with the equipment supplier that an engineer's visit is planned, this can deter this type of attack.
- This approach is much simpler than trying to address the problem using a technological solution. When a new system is to be developed, security risk assessment and management should be a continuing process throughout the development life cycle from initial specification to operational use.
- The stages of risk assessment are:
  - ➢ **Preliminary Risk Assessment:** The aim of this initial risk assessment is to identify generic risks that are applicable to the system and to decide if an adequate level of security can be achieved at a reasonable cost. At this stage, decisions on the detailed system requirements, the system design, or the implementation technology have not been made. The risk assessment should therefore focus on the identification and analysis of high-level risks to the system. The outcomes of the risk assessment process are used to help identify security requirements.
  - ➢ **Design Risk Assessment:** This risk assessment takes place during the system development life cycle and is informed by the technical system design and implementation decisions. The results of the assessment may lead to changes to the security requirements and the addition of new requirements. Known and potential vulnerabilities are identified, and this knowledge is used to inform decision making about the system functionality and how it is to be implemented, tested, and deployed.
  - ➢ **Operational Risk Assessment:** This risk assessment process focuses on the use of the system and the possible risks that can arise. For example, when a system is used in an environment where interruptions are common, a security risk is that a logged-in user leaves his or her computer unattended to deal with a problem. To counter this risk, a timeout requirement may be specified so that a user is automatically logged out after a period of inactivity. Operational risk assessment should continue after a system has been installed to take account of how the system is used and proposals for new and changed requirements.
- Organizational changes may mean that the system is used in different ways from those originally planned.
- The specification of security requirements for systems has much in common with the specification of safety requirements. You cannot specify safety or security requirements as probabilities. Like safety requirements, security requirements are often "shall not" requirements that define unacceptable system behavior rather than required system functionality.
- However, security is a more challenging problem than safety, for a number of reasons:
  - ➢ When considering safety, you can assume that the environment in which the system is installed is not hostile. No one is trying to cause a safety-related incident. When considering security, you have to assume that attacks on the system are deliberate and that the attacker may have knowledge of system weaknesses.
  - ➢ When system failures occur that pose a risk to safety, you look for the errors or omissions that have caused the failure. When deliberate attacks cause system failure, finding the root cause may be more difficult as the attacker may try to conceal the cause of the failure.
  - ➢ It is usually acceptable to shut down a system or to degrade system services to avoid a safety-related

failure. However, attacks on a system may be denial-of service attacks, which are intended to compromise system availability. Shutting down the system means that the attack has been successful.

- Safety-related events are accidental and are not created by an intelligent adversary. An attacker can probe a system's defenses in a series of attacks, modifying the attacks as he or she learns more about the system and its responses. These distinctions mean that security requirements have to be more extensive than safety requirements. Safety requirements lead to the generation of functional system requirements that provide protection against events and faults that could cause safety-related failures.

- These requirements are mostly concerned with checking for problems and taking actions if these problems occur.
- By contrast, many types of security requirements cover the different threats faced by a system. Firesmith (Firesmith 2003) identified 10 types of security requirements that may be included in a system specification:
  - Identification requirements specify whether or not a system should identify its users before interacting with them.
  - Authentication requirements specify how users are identified.
  - Authorization requirements specify the privileges and access permissions of identified users.
  - Immunity requirements specify how a system should protect itself against viruses, worms, and similar threats.
  - Integrity requirements specify how data corruption can be avoided.
  - Intrusion detection requirements specify what mechanisms should be used to detect attacks on the system.
  - Non repudiation requirements specify that a party in a transaction cannot deny its involvement in that transaction.
  - Privacy requirements specify how data privacy is to be maintained.
  - Security auditing requirements specify how system use can be audited and checked.
  - System maintenance security requirements specify how an application can prevent authorized changes from accidentally defeating its security mechanisms.

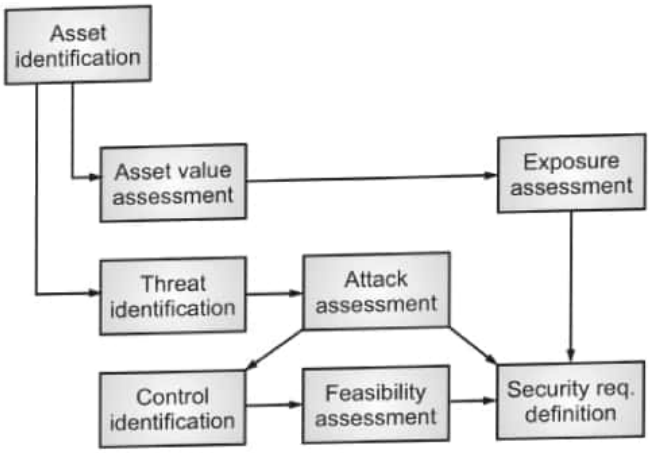


Fig. 6.9 : The preliminary risk assessment process for security requirements

- Risk detection requirements define mechanisms that identify the risk if it arises and neutralize the risk before losses occur.
- Risk mitigation requirements set out how the system should be designed so that it can recover from and restore system assets after some loss has occurred.
- A risk-driven security requirements process is shown in Figure 6.9.
- The process stages are:
  - Asset identification, where the system assets that may require protection are identified. The system itself or particular system functions may be identified as assets as well as the data associated with the system.
  - Asset value assessment, where you estimate the value of the identified assets.
  - Exposure assessment, where you assess the potential losses associated with each asset. This process should take into account direct losses such

as the theft of information, the costs of recovery, and the possible loss of reputation.

- ➢ Threat identification, where you identify the threats to system assets.
- ➢ Attack assessment, where you decompose each threat into attacks that might be made on the system and the possible ways in which these attacks may occur.

6.6 DESIGN GUIDELINES

- Some of the guidelines are as below:

Guideline 1: Base Security Decisions on an Explicit Security Policy

Table 6.5: Design Guidelines for Secure Systems Engineering

Design Guidelines for Security

1	Base security decisions on an explicit security policy
2	Use defense in depth
3	Fail securely
4	Balance security and usability
5	Log user actions
6	Use redundancy and diversity to reduce risk
7	Specify the format of system inputs
8	Compartmentalize your assets
9	Design for development
10	Design for recovery

Guideline 2: Use Defense in Depth

- In any critical system, it is good design practice to try to avoid a single point of failure. That is, a single failure in part of the system should not result in an overall systems failure. In security terms, this means that you should not rely on a single mechanism to ensure security; rather, you should employ several different techniques. This concept is sometimes called "defense in depth."

Guideline 3: Fail Securely

- System failures are inevitable in all systems, and, in the same way that safety-critical systems should always fail-safe; security-critical systems should always "fail-secure." When the system fails, you should not use fallback procedures that are less secure than the system itself. Nor should system failure mean that an attacker can access data that would not normally be allowed.

Guideline 4: Balance Security and Usability

- The demands of security and usability are often contradictory. To make a system secure, you have to introduce checks that users are authorized to use the system.

Guideline 5: Log User Actions

- If it is practically possible to do so, you should always maintain a log of user actions. This log should, at least, record who did what, the assets used and the time and date of the action. If you maintain this as a list of executable commands, you can replay the log to recover from failures. You also need tools that allow you to analyze the log and detect potentially anomalous actions. These tools can scan the log and find anomalous actions, and thus help detect attacks and trace how the attacker gained access to the system

Guideline 6: Use Redundancy and Diversity to Reduce Risk

- Redundancy means that you maintain more than one version of software or data in a system. Diversity, when applied to software, means that the different versions should not rely on the same platform or be implemented using the same technologies. Therefore, platform or technology vulnerabilities will not affect all versions and so will lead to a common failure

Guideline 7: Specify the Format of System Inputs

- A common attack on a system involves providing the system with unexpected inputs that cause it to behave in an unanticipated way. These inputs may simply cause a system crash, resulting in a loss of service, or the inputs could be made up of malicious code that is executed by the system. Buffer overflow vulnerabilities,

first demonstrated in the Internet worm (Spafford 1989) and commonly used by attackers, may be triggered using long input strings. So-called SQL poisoning, where a malicious user inputs an SQL fragment that is interpreted by a server, is another fairly common attack.

Guideline 8: Compartmentalize your Assets

- This means that the effects of an attack that compromises an individual user account may be contained. Some information may be lost or damaged, but it is unlikely that all of the information in the system will be affected.

Guideline 9: Design for Deployment

- Many security problems arise because the system is not configured correctly when it is deployed in its operational environment. Deployment means installing the software . computers where it will execute and setting software parameters to reflect the execution environment and the preferences of the system user.

Guideline 10: Design for Recovery

- Irrespective of how much effort you put into maintaining systems security, you should always design your system with the assumption that a security failure could occur. Therefore, you should think about how to recover from possible failures and restore the system to a secure operational status.

EXERCISE

1. Explain Dependability properties.
2. Explain
   (a) Availability
   (b) Reliability
   (c) Safety
   (d) Security
3. Explain dependability properties.
4. Explain Verification and validation processes.
5. Explain in detail.
   (a) Fault Avoidance
   (b) Fault Detection and Correction
   (c) Fault Tolerance.
6. Explain Reliability Measurement.
7. Explain Safety Terminology.
8. Explain Safety Cases.
9. Explain Design Guidelines.

✠ ✠ ✠

Model Question Papers for End-Semester Examination

PAPER I

Time : 3 Hours **Max. Marks : 60**

Instructions to the candidates :

1. Each Question carries 12 Marks.
2. Attempt any five questions to the following.
3. Illustrate your answers with neat sketches, diagram etc., wherever necessary.
4. If some part or parameter is noticed to be missing, you may appropriately assume it and should mention it clearly.

Attempt any Five Questions

1. Attempt any Two of the Following :

(a) What is software engineering? Briefly discuss the need for software engineering. **[6]**

(b) What are attributes of good software? Explain the key challenges facing software engineering **[6]**

(c) What is software engineering? Explain software engineering code of ethics. **[6]**

2. Attempt any three of the following.

(a) With the help of diagram explain waterfall model. Also state its advantages and disadvantages. **[4]**

(b) What is RAD model- advantages, disadvantages and when to use it? **[4]**

(c) Explain agility and cost of change. **[4]**

(d) State XP practice principles. **[4]**

3. Attempt the following.

(a) What is requirement engineering. Distinguish functional and non functional requirements. **[6]**

(b) Explain Layered information system architecture. **[6]**

4. Attempt any two of the following.

(a) Explain different ways of writing a system requirements specification. **[6]**

(b) Draw a sequence diagram showing the interactions of objects in a group diary system when a group of people are arranging a meeting. **[6]**

(c) With the help of neat diagram explain six stages of Acceptance testing process. **[6]**

5. Attempt any two of the following.

(a) Draw a UML state diagram showing the possible state changes in either the group diary or the filling station system. **[6]**

(b) What is the release testing? What are the differences between release testing and system testing? **[6]**

(c) What is performance testing? Give examples of performance test cases. **[6]**

6. Attempt the following.

(a) Explain in detail. **[6]**

(i) Fault Avoidance (ii) Fault Detection and Correction

(b) Write a scenario that could be used to help design tests for the wilderness weather station system. **[6]**

✠ ✠ ✠

PAPER II

Time : 3 Hours **Max. Marks : 60**

Instructions to the candidates :

1. Each Question carries 12 Marks.
2. Attempt any five questions to the following.
3. Illustrate your answers with neat sketches, diagram etc., wherever necessary.
4. If some part or parameter is noticed to be missing, you may appropriately assume it and should mention it clearly.

Attempt any Five Questions

1. Attempt any Two of the Following :

(a) What is ethics doing in a course for software engineers? Explain with the help of case study. **[6]**

(b) Explain model for Mental Health Care-Patient Management System. **[6]**

(c) Design a model to help monitor climate change and to improve the accuracy of weather forecast. **[6]**

2. Attempt any two of the following.

(a) What is a software process model? Explain the types of software process models. **[6]**

(b) What are industrial XP practices? Explain. **[6]**

(c) Explain Scrum Process Flow with neat diagram. **[6]**

3. Attempt the following.

(a) How the agile methods are scaled? State the coping of agile methods for large system engineering. **[6]**

(b) Draw Sequence diagrams for ATM system. **[6]**

4. Attempt the following.

(a) Draw State transiton diagram for ATM system. **[6]**

(b) Explain different ways of writing a system requirements specification. **[6]**

5. Attempt any two of the following.

(a) Draw a UML state diagram showing the possible state changes in either the group diary or the filling station system. **[6]**

(b) Explain Test driven development with neat diagram. Also mention its advantages. **[6]**

(c) What is acceptance testing? Why it is important? **[6]**

6. Attempt the following.

(a) Explain (i) Availability (ii) reliability (iii) Safety **[6]**

(b) Explain in detail.

(i) Fault Avoidance (ii) Fault Detection and Correction **[6]**

✠ ✠ ✠