SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-I- SOFTWARE ENGINEERING– SBS1204

SBS1204 - SOFTWARE ENGINEERING

UNIT 1

Definition of software and software engineering – Software myths –Software Engineering paradigms: Linear Sequential Model and Prototyping Model-Incremental – Spiral Model-Iterative Model. Software Project Management. Software Cost Estimation – Software Project Planning.

SOFTWARE is

(1) Instructions (computer programs) that when executed provide desired function and performance,

(2) Data structures that enable the programs to adequately manipulate information, and

(3) Documents that describe the operation and use of the programs.

SOFTWARE APPLICATIONS

System software. System software is a collection of programs written to service other programs. Some system software

e.g., compilers, editors, and file management utilities

Real-time software. Software that monitors/analyzes/controls real-world events as they occur is called real time

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.

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.

Embedded software. Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets.

Personal computer software. The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

Web-based software. The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats).

Artificial intelligence software. Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.

SOFTWARE MYTHS

Many software problems arise due to myths that are formed during the initial stages of software development. Unlike ancient folklore that often provides valuable lessons, software myths propagate false beliefs and confusion in the minds of management, users and developers.

Management myths

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?

Myth: If we get behind schedule, we can add more programmers and catch up

Reality: Software development is not a mechanistic process like manufacturing. adding people to a late software project makes it later." However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.

Myth: If I decide to outsource3 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.

Customer myths

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

Reality: 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 because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. If serious attention is given to up-front definition, early requests for change can be accommodated easily.

Practitioner's myths

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of 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 configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

SOFTWARE ENGINEERING

- Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines

- Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

- software engineering is a discipline whose aim is the production of fault-free software, delivered on time and within budget, that satisfies the client's needs. Furthermore, the software must be easy to modify when the user's needs change.

SOFTWARE ENGINEERING PARADIGM

Software engineering paradigm is refers to the method and steps, which are taken while design the software. It consists of three parts.

- Software development paradigm
- Software design paradigm
- Programming paradigm

Software development paradigm

Software development paradigm pertains all the engineering concepts which includes Requirements, software design, programming.

Software design paradigm

Software design paradigm is a part of software development . it includes design, programming, maintenance.

Programming paradigm

Programming paradigm which concerns about the programming aspect of software development. This includes coding, testing, integration.

Need for software engineering

Software engineering fulfill the higher rate of change in user requirement and environment. The user requirement lies on cost, scalability, dynamaic nature of working environments, quality management.

Characteristics of good software

The software product can be judged by what it offer and how well it can be used . furthermore, the software must satisfied on the following grounds :

- Operational
- Transitional
- Maintainance

a) Operational

Operational defines how well the software works in operations. It can be measure on budget, usability, efficiency, correctness, functionality, dependability, security, safety.

b) Transitional

This aspect is important if the software is moved from one platform to another.

The key parameters that includes portability, Interoperability, Reusability, Adaptability.

c) Maintenance

Maintenance of the software defines the capabilities to maintain itself in the ever changing environment. It includes modularity, maintainability, flexibility etc.

SOFTWARE DEVELOPMENT LIFE-CYCLE MODELS

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process. The following figure is a graphical representation of the various stages of a typical SDLC. The SDLC consist of the following phases:

- Communication

- Requirement gathering
- Feasibility study
- System Analysis
- Software design
- Coding
- Testing and Integration
- Operations & maintenance
- Disposition

Process framework: activities

The followings are the key activities of process models:

- Communication: communication and collaboration between developer and customer or stakeholder.
- Planning: technical task to be conducted, risk analysed, resources required, work schedule.
- Modeling : creation of model for better understanding
- Construction : combination of code generation and testing.
- Deployment software is delivered to customers and getting feedback.

Code-and-Fix Life-Cycle Model

- The product is implemented without requirements or specifications, or any attempt at design. Instead, the developers simply throw code together and rework it as many times as necessary to satisfy the client
- Although this approach may work well on short programming exercises 100 or 200 lines long, the code-and-fix model is totally unsatisfactory for products of any reasonable size.
- The cost of the code-and-fix approach is actually far greater because the change is done only after the coding is completed.

- Maintenance of a product can be extremely difficult without specification or design documents, and the chances of a regression fault occurring are considerably greater.

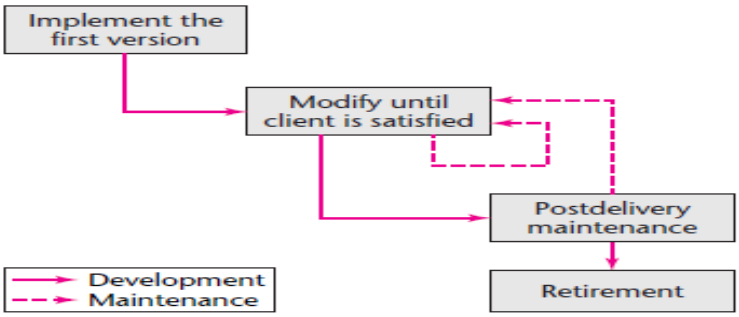
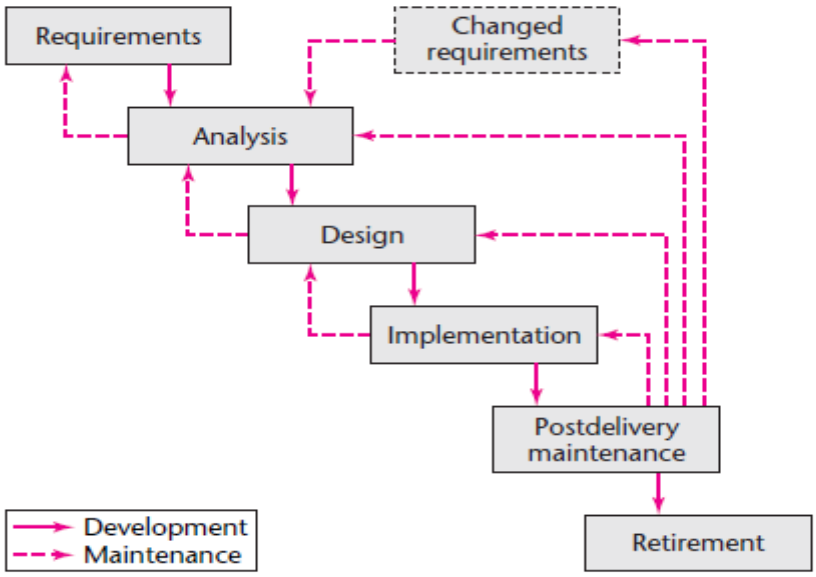


Fig.1.1. Code and Fix Life cycle model

Waterfall Life-Cycle Model



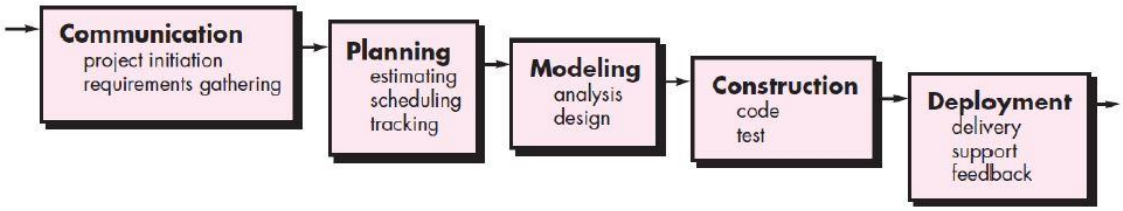


Fig.1.2.Waterfall Life Cycle model

- The classic life cycle suggests a systematic, sequential approach to software development.
- A critical point regarding the waterfall model is that no phase is complete until the documentation for that phase has been completed and the products of that phase have been approved by the software quality assurance (SQA) group.
- Inherent in every phase of the waterfall model is testing. Testing is not a separate phase to be performed only after the product has been constructed, nor is it to be performed only at the end of each phase. Instead, testing should proceed continually throughout the software process.
- In particular, during maintenance, it is necessary to ensure not only that the modified version of the product still does what the previous version did—and still does it correctly (regression testing)—but that it also satisfies any new requirements imposed by the client.
- The feedback loops permits modifications to be made to design documents, the software project management plan, and even the specification document, if necessary.
- The specification document, design document , code document and other documents such as database manual, user manual and operational manual are essential tool for maintaining the product.

Advantage

- Easy to understand and implement.
- Widely used and known

- Reinforces good habits: define-before- design, design-before-code
- Identifies deliverables and milestones
- Document driven
- Maintenance is easier

Disadvantage

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.
- Delivered product may not meet client needs

Evolutionary Software Process Models

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

Incremental Model

- Software is constructed step by step , in the same way that a building is constructed
- Incremental model in software engineering is a one which combines the elements of waterfall model which are then applied in an iterative manner. It basically delivers a series of releases called increments which provide progressively more functionality for the client as each increment is delivered.
- In incremental model of software engineering, waterfall model is repeatedly applied in each increment. The incremental model applies linear sequences in a required pattern as calendar time passes. Each linear sequence produces an increment in the work.
- When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered.
- The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.

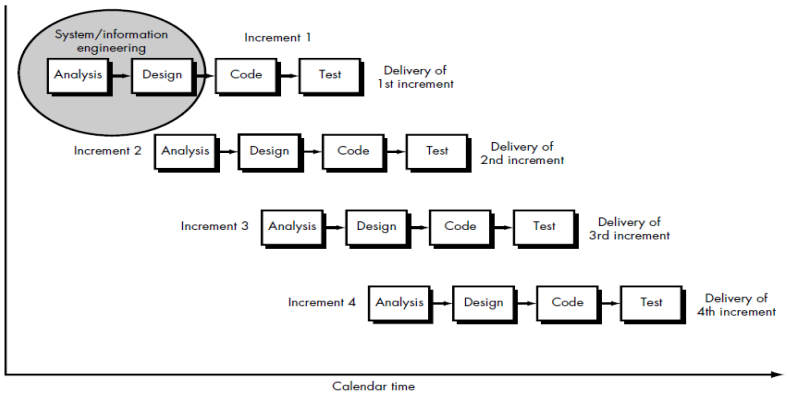


Fig.1.3. Incremental Life Cycle model

Advantages Of Incremental Model

- Initial product delivery is faster.
- Lower initial delivery cost.
- Core product is developed first i.e main functionality is added in the first increment.
- After each iteration, regression testing should be conducted. During this testing, faulty elements of the software can be quickly identified because few changes are made within any single iteration.
- It is generally easier to test and debug than other methods of software development because relatively smaller changes are made during each iteration. This allows for more targeted and rigorous testing of each element within the overall product.
- With each release a new feature is added to the product.
- Customer can respond to feature and review the product.
- Risk of changing requirement is reduced
- Work load is less.

Disadvantages Of Incremental Model

- Needs good planning and design.
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
- Total cost is higher than waterfall.

- Mostly such model is used in web applications and product based companie

Spiral Model

- An element of risk is always involved in the development of software.
- For example, key personnel can resign before the product has been adequately documented. Too much, or too little, can be invested in testing and quality assurance. After spending hundreds of thousands of dollars on developing a major software product, technological breakthroughs can render the entire product worthless.
- An organization may research and develop a database management system, but before the product can be marketed, a lower-priced, functionally equivalent package is announced by a competitor.
- For obvious reasons, software developers try to minimize such risks wherever possible. One way of minimizing certain types of risk is to construct a prototype.
- The idea of minimizing risk via the use of prototypes and other means is the idea underlying the spiral life-cycle model. A simplified way of looking at this lifecycle model is as a waterfall model with each phase preceded by risk analysis.

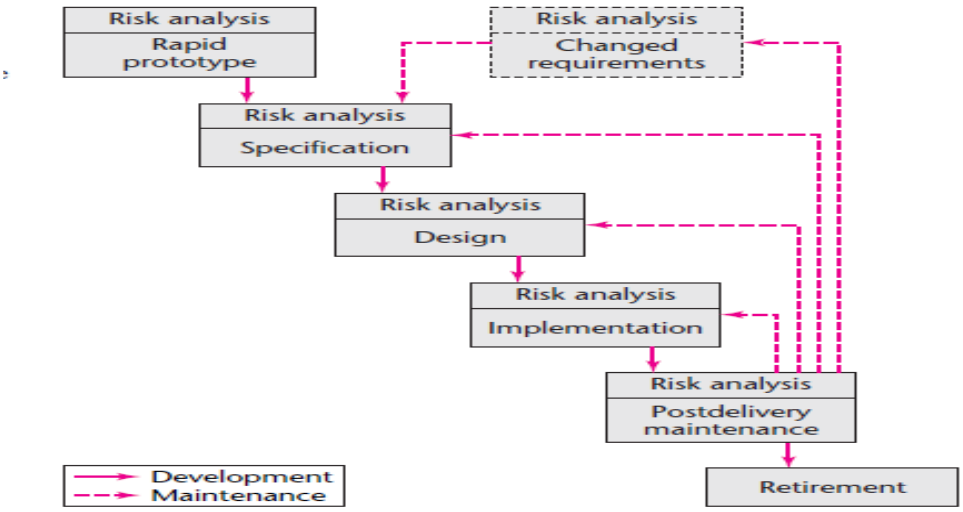


Fig.1.4. A simplified version of the spiral life-cycle model.

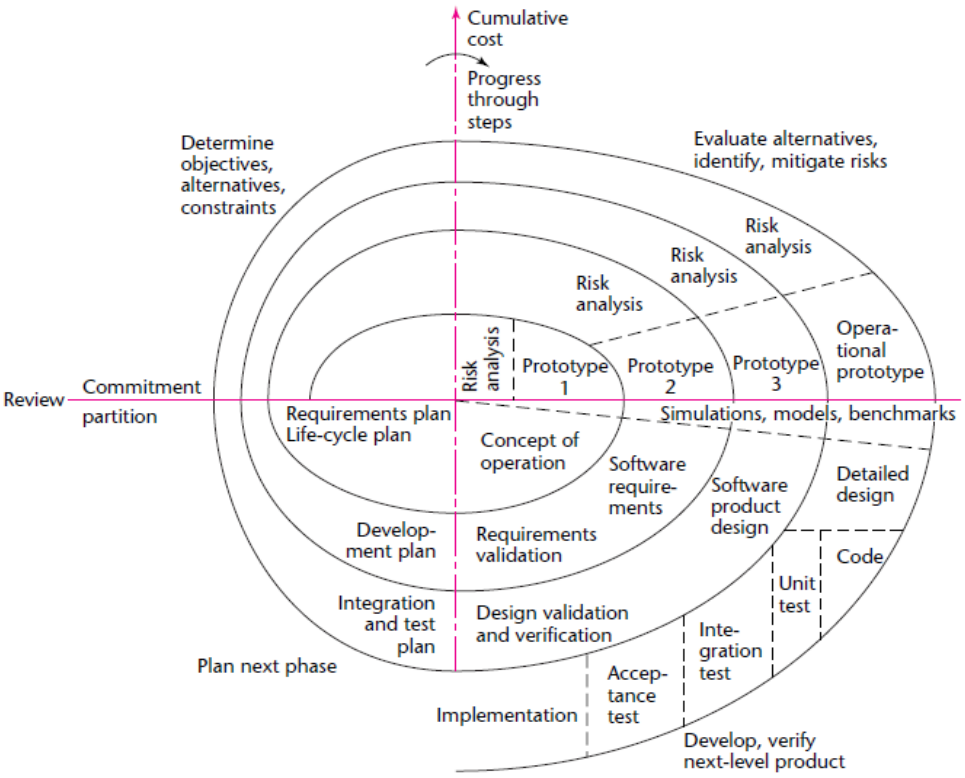


Fig.1.5.Full spiral life-cycle model

- In full spiral model the radial dimension represents cumulative cost to date, and the angular dimension represents progress through the spiral.
- Each cycle of the spiral corresponds to a phase. A phase begins (in the top left quadrant) by determining objectives of that phase, alternatives for achieving those objectives, and constraints imposed on those alternatives. This process results in a strategy for achieving those objectives.
- Attempts are made to mitigate every potential risk, in some cases by building a prototype.
- If certain risks cannot be mitigated, the project may be terminated immediately; under some circumstances, however, a decision could be made to continue the project but on a significantly smaller scale.

- If all risks are successfully mitigated, the next development step is started (bottom right quadrant). This quadrant of the spiral model corresponds to the classical waterfall model. Finally, the results of that phase are evaluated and the next phase is planned.

Advantages of Spiral model

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.
- Project estimates in terms of schedule, cost etc become more and more realistic as the project moves forward and loops in spiral get completed.
- It is suitable for high risk projects, where business needs may be unstable. A highly customized product can be developed using this.

Disadvantages of Spiral model

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.
- It is not suitable for low risk projects.
- May be hard to define objective, verifiable milestones.
- Spiral may continue indefinitely.

Rapid-Prototyping Life-Cycle Model

- A rapid prototype is a working model that is functionally equivalent to a subset of the product. For example, if the target product is to handle accounts payable, accounts receivable, and warehousing, then the rapid prototype might consist of a product that performs the screen handling for data capture and prints the reports, but does no file updating or error handling.

- The first step in the rapid-prototyping life-cycle model is to build a rapid prototype and let the client and future users interact and experiment with the rapid prototype. Once the client is satisfi ed that the rapid prototype indeed does most of what is required, the developers can draw up the specification document with some assurance that the product meets the client's real needs.

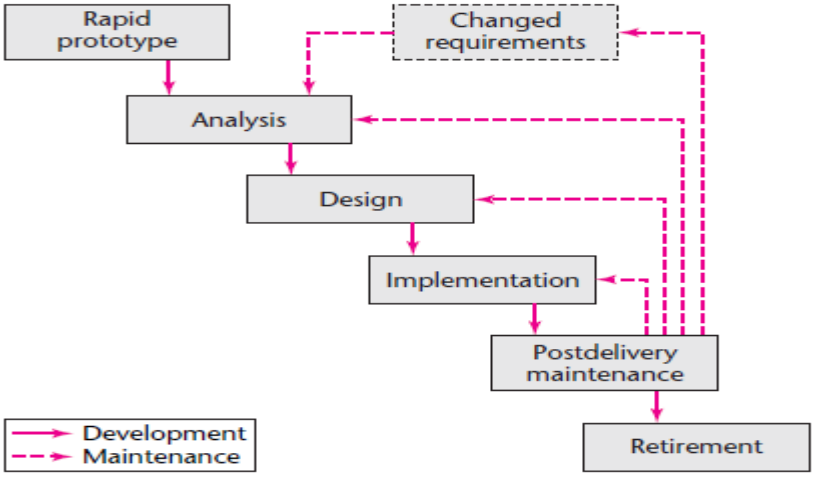


Fig.1.6.The rapidprototyping life cycle model

- A major strength of the rapid-prototyping model is that the development of the product is essentially linear, proceeding from the rapid prototype to the delivered product; the feedback loops of the waterfall model are less likely to be needed in the rapid-prototyping model.

Advantages

- Provides a working model to the user early in the process , enabling early assessment and increasing user confidence.
- The developer gains experience and insight by developing a prototype , thereby resulting in better implementation of requirements.
- Helps in reducing risks associated with the project.
- The prototyping model serves to clarify requirements , which are not clear , hence reducing ambiguity and improving communication between the developer and the user.

- There is a great involvement of users in software development . Hence , the requirement of the users are met to the greatest extent.

Disadvantages

- If the user is not satisfied with the developed prototype, then a new prototype is developed . This process goes on until a perfect prototype evolves . Thus , this model is time consuming and expensive.
- The developer loses focus of the real purpose of prototype and compromises on the quality of the product . For example , he may apply some of the inefficient algorithms or inappropriate programming languages used in developing the prototype .
- Prototyping can lead to false expectations. It often creates a situation where the user belives that the development of the system is finished when it is not.
- The primary goal of prototyping is rapid development. Thus , the design of the system may suffer as it is built in a series of layers without considering integration of all the other components.

Object-Oriented Life-Cycle Models

- Need for iteration within and between phases
  - Fountain model
  - Unified software development process
- All incorporate some form of
  - Iteration
  - Parallelism
  - Incremental development

Fountain Model

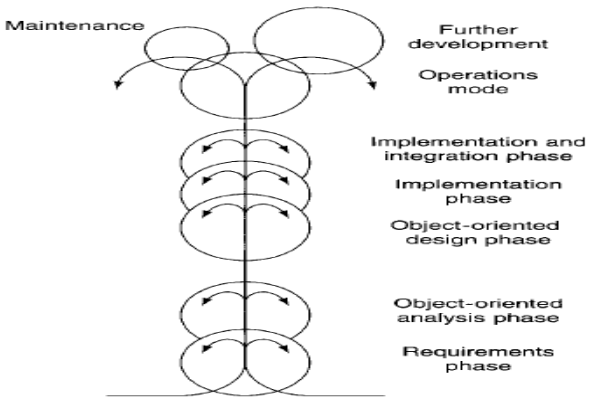


Fig.1.7.Fountain Model

- Objetct oriented life cycle model have been proposed that explicitily reflect the need for iteration
- The circles representing the various phases overlap,explicitly reflecting an overlap between activities.
- The arrows within a phase represent iteration within that phase.
- The maintenance circle is smaller , to symbolize reduce maintenance effort when the object oriented paradigm is used.

Advantages

- Support Iteration within phases
- Parallelism between phases

Disadvantages

- It may be degraded in to CABTAB(code-a-bit test-a-bit) which reuires frequent itreation and refinements

Unified Process

- Unified process is a framework for OO software engineering using UML (Unified Modeling Language)
- Unified process (UP) is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them in a way that implements many of the best principles of agile software development

Inception phase

- Encompasses the customer communication and planning activities
- Rough architecture, plan, preliminary use-cases

Elaboration phase

- Encompasses the customer communication and modeling activities
- Refines and expands preliminary use-cases
- Expands architectural representation to include: use-case model, analysis model, design model, implementation model, and deployment model
- The plan is carefully reviewed and modified if needed

Construction phase

- Analysis and design models are completed to reflect the final version of the software increment
- Using the architectural model as an input develop or acquire the software components, unit tests are designed and executed, integration activities are conducted
- Use-cases are used to derive acceptance tests

Transition phase

- Software is given to end-users for beta testing
- User report both defects and necessary changes
- Support information is created (e.g., user manuals, installation procedures)
- Software increment becomes usable software release

• Production phase

- Software use is monitored
- Defect reports and requests for changes are submitted and evaluated

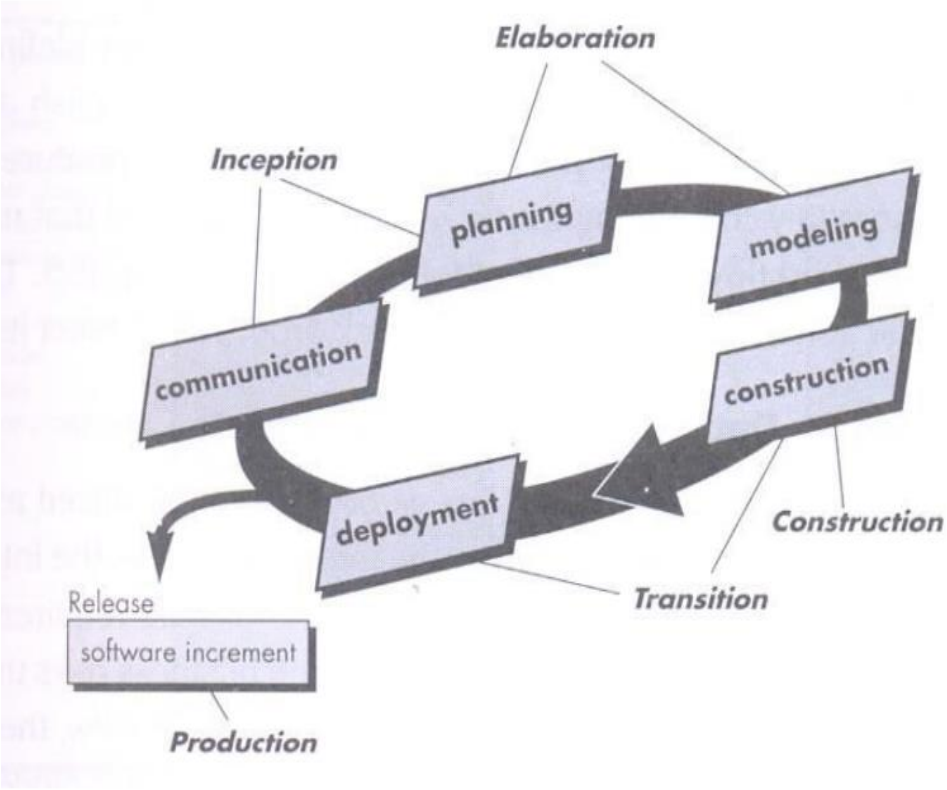


Fig.1.8. Unified Process Model

UNIFIED PROCESS WORK PRODUCTS

• Tasks which are required to be completed during different phases

• **Inception Phase**
  - Vision document
  - Initial Use-Case model
  - Initial Risk assessment
  - Project Plan

• **Elaboration Phase**
  - Use-Case model
  - Analysis model
  - Software Architecture description
  - Preliminary design model

• **Construction Phase**
  - Design model
  - System components
  - Test plan and procedure
  - Test cases
  - Manual

• **Transition Phase**
  - Delivered software increment
  - Beta test results
  - General user feedback

Verification and Validation

- **Validation**: Are we building the right system?
- **Verification**: Are we building the system right?

- validation is concerned with checking that the system will meet the customer's actual needs, verification is concerned with whether the system is well-engineered, error-free, and so on.
- Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful.
- The distinction between the two terms is largely to do with the role of specifications. Validation is the process of checking whether the specification captures the customer's

needs, while verification is the process of checking that the software meets the specification.

- Verification includes all the activities associated with the producing high quality software: testing, inspection, design analysis, specification analysis, and so on.
- Validation includes activities such as requirements modelling, prototyping and user evaluation.
- In a traditional phased software lifecycle, verification is often taken to mean checking that the products of each phase satisfy the requirements of the previous phase.
- Validation is relegated to just the begining and ending of the project: requirements analysis and acceptance testing. This view is common in many software engineering textbooks, and is misguided. It assumes that the customer's requirements can be captured completely at the start of a project, and that those requirements will not change while the software is being developed. In practice, the requirements change throughout a project, partly in reaction to the project itself: the development of new software makes new things possible. Therefore both validation and verification are needed throughout the lifecycle.

SOFTWARE PROJECT MANAGEMENT

Effective software project management focuses on the four P's: people, product, process, and project.

The People

The "people factor" is so important that the Software Engineering Institute has developed a people management capability maturity model (PM-CMM), "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability".

The people management maturity model defines the following key practice areas for software people: recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development.

The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified.

The software developer and customer must meet to define product objectives and scope. Objectives identify the overall goals for the product (from the customer's point of view) without considering how these goals will be achieved. Scope identifies the primary data, functions and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner.

Once the product objectives and scope are understood, alternative solutions are considered.

The Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, 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. Finally, umbrella activities—such as software quality assurance, software configuration management,and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project

management, and develop a commonsense approach for planning, monitoring and controlling the project.

People

we examine the players who participate in the software process and the manner in which they are organized to perform effective software engineering.

The Players

The software process (and every software project) is populated by players who can be categorized into one of five constituencies:

1. Senior managers who define the business issues that often have significant influence on the project.
2. Project (technical) managers who must plan, motivate, organize, and control the practitioners who do software work.
3. Practitioners who deliver the technical skills that are necessary to engineer a product or application.
4. Customers who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
5. End-users who interact with the software once it is released for production use.

Team Leaders

Project management is a people-intensive activity, and for this reason, competent practitioners often make poor team leaders.

In an excellent book of technical leadership, Jerry Weinberg suggests a MOI model of leadership:

Motivation. The ability to encourage technical people to produce to their best ability.

Organization. The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation. The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time, letting everyone on the team know that quality counts and that it will not be compromised.

Another view of the characteristics that define an effective project manager emphasizes four key traits:

Problem solving. An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity. A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement. To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actions that controlled risk taking will not be punished.

Influence and team building. An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

The Software Team

The "best" team structure depends on the management style of your organization,the number of people who will populate the team and their skill levels, and the overall problem difficulty.

Mantei describes seven project factors that should be considered when planning the structure of software engineering teams:

• The difficulty of the problem to be solved.

• The size of the resultant program(s) in lines of code or function points

• The time that the team will stay together (team lifetime).
• The degree to which the problem can be modularized.
• The required quality and reliability of the system to be built.
• The rigidity of the delivery date.
• The degree of sociability (communication) required for the project

To achieve a high-performance team:
• Team members must have trust in one another.
• The distribution of skills must be appropriate to the problem.
• Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.

Coordination and Communication Issues

Kraul and Streeter examine a collection of project coordination techniques that are categorized in the following manner:

Formal, impersonal approaches include software engineering documents and deliverables (including source code), technical memos, project milestones, schedules, and project control tools, change requests and related documentation, error tracking reports, and repository data .

Formal, interpersonal procedures focus on quality assurance activities applied to software engineering work products. These include status review meetings and design and code inspections.

Informal, interpersonal procedures include group meetings for information dissemination and problem solving and "collocation of requirements and development staff."

Electronic communication encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.

Interpersonal networking includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

THE PRODUCT

Software Scope

The first software project management activity is the determination of software scope. Scope is defined by answering the following questions:

Context. How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?

Information objectives. What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance. What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Problem Decomposition

Problem decomposition, sometimes called partitioning or problem elaboration, is an activity that sits at the core of software requirements analysis. During the scoping activity no attempt is made to fully decompose the problem. Rather,decomposition is applied in two major areas: (1) the functionality that must be delivered and (2) the process that will be used to deliver it.

THE PROCESS

The problem is to select the process model that is appropriate for the software to be engineered by a project team. software engineering paradigms are

- the linear sequential model
- the prototyping model
- the RAD model
- the incremental model
- the spiral model
- the WINWIN spiral model
- the component-based development model
- the concurrent development model
- the formal methods model
- the fourth generation techniques model

Melding the Product and the Process

Project planning begins with the melding of the product and the process. Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization. Assume that the organization has adopted the following set of framework activities

• **Customer communication**—tasks required to establish effective requirements elicitation between developer and customer.

• **Planning**—tasks required to define resources, timelines, and other projectrelated information.

• **Risk analysis**—tasks required to assess both technical and management risks.

• **Engineering**—tasks required to build one or more representations of the application.

• **Construction and release**—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

• **Customer evaluation**—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering activity and implemented during the construction activity.

Process Decomposition

Process decomposition commences when the project manager asks, "How do we accomplish this common process framework(CPF) activity?" For example, a small,relatively simple project might require the following work tasks for the customer communication activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

Now, we consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the customer communication activity:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software.
7. Review each mini-spec for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document with all concerned.
10. Modify the scoping document as required.

THE PROJECT

In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right. John Reel defines ten signs that indicate that an information systems project is in jeopardy:

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4.The chosen technology changes.
5. Business needs change [or are ill-defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned

But enough negativity! How does a manager act to avoid the problems just noted? Reel suggests a five-part commonsense approach to software projects:

1. Start on the right foot. This is accomplished by working hard (very hard) to understand the problem that is to be solved . It is reinforced by building the right team and giving the team the autonomy, authority, and technology needed to do the job.

2. Maintain momentum. Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs.

3. Track progress. For a software project, progress is tracked as work products (e.g., specifications, source code, sets of test cases) are produced and approved (using formal technical reviews) as part of a quality assurance activity.

4.Make smart decisions. In essence, the decisions of the project manager and the software team should be to "keep it simple."

5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

SOFTWARE PROJECT PLANNING

Software Project Planning atually encompasses planning involves estimation—your attempt to determine how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product.

PROJECT PLANNING OBJECTIVES

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses.

SOFTWARE SCOPE

The first activity in software project planning is the determination of software scope. A statement of software scope must be bounded.Software scope describes the data and control to be processed, function, performance, constraints, interfaces, and reliability. Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful.

Obtaining Information Necessary for Scope

The most commonly used technique to bridge the communication gap between the customer and developer and to get the communication process started is to conduct a preliminary meeting or interview.Gause and Weinberg suggest that the analyst start by asking context-free questions; that is, a set of questions that will lead to a basic understanding of the problem,For example, the analyst might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution?

The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice any perceptions about a solution:

- How would you (the customer) characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Will any special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the meeting. Gause and Weinberg call these "meta-questions" and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Feasibility

Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: “Can we build software to meet this scope? Is the project feasible?”

A feasibility study decides whether or not the proposed system is worthwhile

- A short focused study that checks
- If the system contributes to organisational objectives;
- If the system can be engineered using current technology and within budget;
- If the system can be integrated with other systems that are used.

- In a feasibility study we need to concentrate our attention on four primary areas of interest:

1. Economic feasibility. An evaluation of development cost weighed against the ultimate income or benefit derived from the developed system or product.
2. Technical feasibility. A study of function, performance, and constraints that may affect the ability to achieve an acceptable system.
3. Legal feasibility. A determination of any infringements, violation, or liability that could result from development of the system.
4. Alternatives. An evaluation of alternative approaches to the development of the system or product.

A Scoping Example

As an example, consider

software for a conveyor line sorting system (CLSS). The statement of scope for CLSS follows:

The conveyor line sorting system (CLSS) sorts boxes moving along a conveyor line. Each box is identified by a bar code that contains a part number and is sorted into one of six bins at the end of the line. The boxes pass by a sorting station that contains a bar code reader and a PC. The sorting station PC is connected to a shunting mechanism that sorts the boxes into the bins. Boxes pass in random order and are evenly spaced. The line is moving at five feet per minute. CLSS is depicted schematically in Figure.

CLSS software receives input information from a bar code reader at time intervals that conform to the conveyor line speed. Bar code data will be decoded into box identification format. The software will do a look-up in a part number database containing a maximum of 1000 entries to determine proper bin location for the box currently at the reader (sorting station). The proper bin location is passed to a sorting shunt that will position boxes in the appropriate bin. A record of the bin destination for each box will be maintained for later recovery and reporting. CLSS software will also receive input from a pulse tachometer that will be used to synchronize the control signal to the shunting mechanism. Based on the number of pulses generated between the sorting station and the shunt, the software will produce a control signal to the shunt to properly position the box.

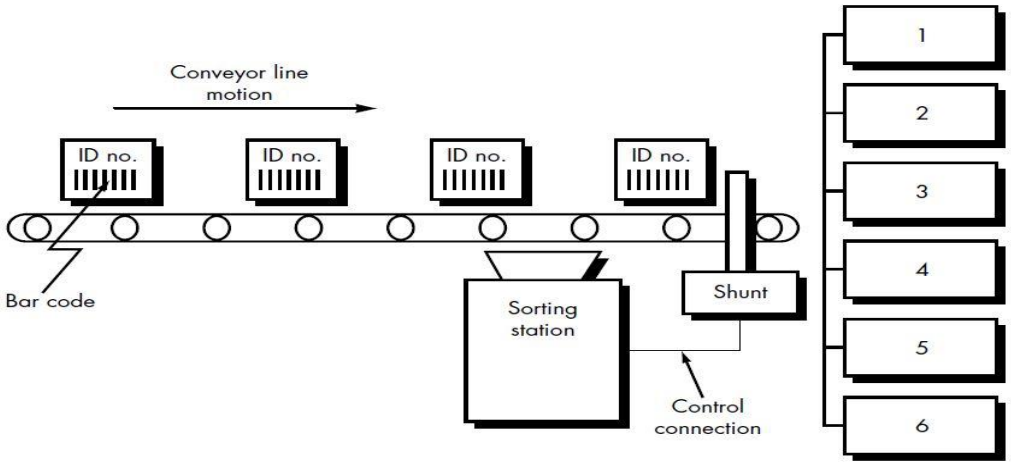


Fig.1.9. Conveyor Line Sorting System (CLSS)

The project planner examines the statement of scope and extracts all important software functions. This process, called decompositionresults in the following functions:4

- Read bar code input.
- Read pulse tachometer.
- Decode part code data.
- Do database look-up.
- Determine bin location.
- Produce control signal for shunt.
- Maintain record of box destinations.

Resources

The second software planning task is estimation of the resources required to accomplish the software development effort.

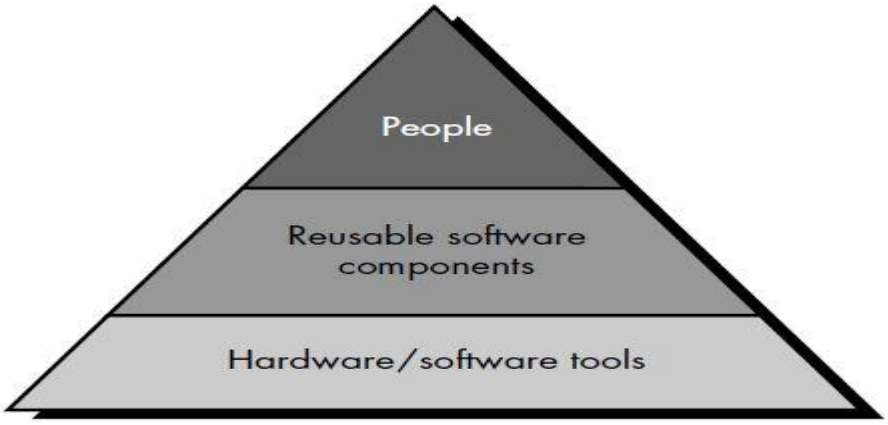


Fig.1.10.Project Resources

Human Resources

The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client/server) are specified. The number of people

required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made.

Reusable Software Resources

Component-based software engineering emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components. Existing software that can be acquired from a third party or that has been developed internally for a past project. are purchased from a third party, are ready for use on the current project, and have been fully validated.

Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components.

Partial-experience components. Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components.

New components. Software omponents that must be built by the software team specifically for the needs of the current project.

Environmental Resources

The environment that supports the software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice

SOFTWARE COST ESTIMATION

Predicting the resources required for a software development process.

Fundamental estimation questions
a) How much effort is required to complete an activity?
b) How much calendar time is needed to complete an activity?
c) What is the total cost of an activity?
d) Project estimation and scheduling are interleaved management activities.

The cost in a project is due to:
a. due the requirements for software, hardware and human resources
b. the cost of software development is due to the human resources needed
c. most cost estimates are measured in person-months (PM)

Software cost components
▪ Hardware and software costs
▪ Travel and training costs
▪ Effort costs (the dominant factor in most projects)
- salaries of engineers involved in the project
- Social and insurance costs

▪ Effort costs must take overheads into account
- costs of building, heating, lighting
- costs of networking and communications
- costs of shared facilities (e.g library, staff restaurant, etc.)

Costing and pricing
▪ Estimates are made to discover the cost, to the developer, of producing a software system
▪ There is not a simple relationship between the development cost and the price charged to the customer
▪ Broader organisational, economic, political and business considerations influence the price charged

Programmer productivity

▪ A measure of the rate at which individual engineers involved in software development produce software and associated documentation.

▪ Not quality-oriented although quality assurance is a factor in productivity assessment

▪ Essentially, we want to measure useful functionality produced per time unit Productivity measures

▪ Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.

▪ Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure

Measurement problems

▪ Estimating the size of the measure

▪ Estimating the total number of programmer months which have elapsed

▪ Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate

Lines of code

▪ What's a line of code?

o The measure was first proposed when programs were typed on cards with one line per card

o How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line

▪ What programs should be counted as part of the system?

▪ Assumes linear relationship between system size and volume of documentation

Function points

▪ Based on a combination of program characteristics

- external inputs and outputs
- user interactions
- external interfaces
- files used by the system

▪ A weight is associated with each of these

The function point count is computed by multiplying each raw count by the weight and summing all values

Object points

▪ Object points are an alternative function-related measure to function points

▪ Object points are NOT the same as object classes
▪ The number of object points in a program is a weighted estimate of
- The number of separate screens that are displayed
- The number of reports that are produced by the system
- The number of modules that must be developed

Productivity estimates
▪ Real-time embedded systems, 40-160 LOC/P-month
▪ Systems programs , 150-400 LOC/P-month
▪ Commercial applications, 200-800 LOC/P-month
▪ In object points, productivity has been measured between 4 and 50 object points/month depending on tool support and developer capability

Quality and productivity
▪ All metrics based on volume/unit time are flawed because they do not take quality into account
▪ Productivity may generally be increased at the cost of quality
▪ It is not clear how productivity/quality metrics are related
▪ If change is constant then an approach based on counting lines of code is not meaningful

Estimation techniques
▪ There is no simple way to make an accurate estimate of the effort required to develop a software system
- Initial estimates are based on inadequate information in a user requirements definition
- The software may run on unfamiliar computers or use new technology
- The people in the project may be unknown

Project cost estimates may be self-fulfilling
- The estimate defines the budget and the product is adjusted to meet the budget

▪ Algorithmic cost modelling
▪ Expert judgement
▪ Estimation by analogy
▪ Parkinson's Law
▪ Pricing to win

Algorithmic code modelling
A formulaic approach based on historical cost information and which is generally based on the size of the software

Expert judgement
▪ One or more experts in both software development and the application domain use their experience to predict software costs. Process iterates until some consensus is reached.
▪ Advantages: Relatively cheap estimation method. Can be accurate if experts have direct experience of similar systems
▪ Disadvantages: Very inaccurate if there are no experts!

Estimation by analogy
▪ The cost of a project is computed by comparing the project to a similar project in the same application domain
▪ Advantages: Accurate if project data available
▪ Disadvantages: Impossible if no comparable project has been tackled. Needs systematically maintained cost database

Parkinson's Law
▪ The project costs whatever resources are available
▪ Advantages: No overspend
▪ Disadvantages: System is usually unfinished
▪ PL states that work expands to fill the time available. The cost is determined by available resources rather than by objective statement.
▪ Example: Project should be delivered in 12 months and 5 people are available.
Effort = 60 p/m

Pricing to win
▪ The project costs whatever the customer has to spend on it
▪ Advantages: You get the contract
▪ Disadvantages: The probability that the customer gets the system he or she wants is small. Costs do not accurately reflect the work required

Top-down and bottom-up estimation
▪ Any of these approaches may be used top-down or bottom-up.
▪ Top-down

- Start at the system level and assess the overall system functionality and how this is delivered through sub-systems.

▪ Bottom-up

- Start at the component level and estimate the effort required for each component. Add these efforts to reach a final estimate.

Top-down estimation

▪ Usable without knowledge of the system architecture and the components that might be part of the system.

▪ Takes into account costs such as integration, configuration management and documentation.

▪ Can underestimate the cost of solving difficult low-level technical problems.

Bottom-up estimation

▪ Usable when the architecture of the system is known and components identified.

▪ This can be an accurate method if the system has been designed in detail.

▪ It may underestimate the costs of system level activities such as integration and documentation.

The COCOMO model

• The COstructive COst Model (COCOMO) is the most widely used software estimation model in the world. It

• The COCOMO model predicts the effort and duration of a project based on inputs relating to the size of the resulting systems and a number of "cost drives" that affect productivity.

▪ An empirical model based on project experience.

▪ Well-documented, 'independent' model which is not tied to a specific software vendor.

▪ Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.

▪ COCOMO 2 takes into account different approaches to software development, reuse, etc.

Effort

• Effort Equation

– PM = C * (KDSI)n (person-months)

• where PM = number of person-month (=152 working hours),

C = a constant,

• KDSI = thousands of "delivered source instructions" (DSI) and

• n = a constant.

Productivity
• Productivity equation
– (DSI) / (PM)
• where PM = number of person-month (=152 working hours),
• DSI = "delivered source instructions"

Schedule
• Schedule equation
– TDEV = C * (PM)n (months)
• where TDEV = number of months estimated for software development.
Average Staffing
• Average Staffing Equation
– (PM) / (TDEV) (FSP)
• where FSP means Full-time-equivalent Software Personnel.

COCOMO Models
• COCOMO is defined in terms of three different models:
– the Basic model,
– the Intermediate model, and
– the Detailed model.
• The more complex models account for more factors that influence software projects, and make more accurate estimates.

The Development mode
• the most important factors contributing to a project's duration and cost is the Development Mode
• **Organic Mode:** The project is developed in a familiar, stable environment, and the product is similar to previously developed products. The product is relatively small, and requires little innovation.
• **Semidetached Mode:** The project's characteristics are intermediate between Organic and Embedded.
• **Embedded Mode:** The project is characterized by tight, inflexible constraints and interface requirements. An embedded mode project will require a great deal of innovation.
Cost Estimation Process
Cost=SizeOfTheProject x Productivity

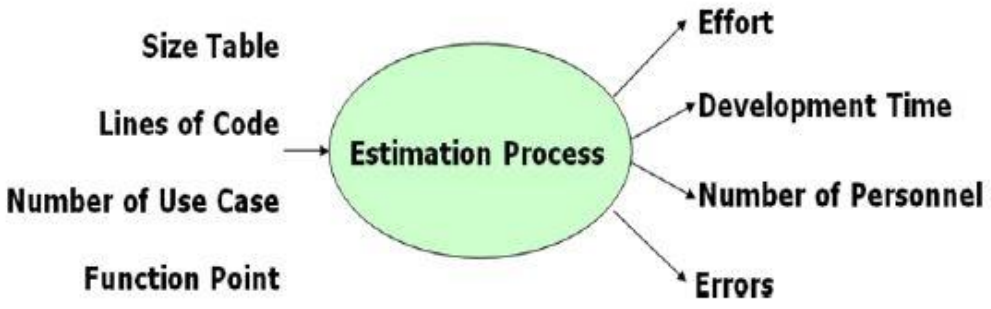


Fig.1.11.Estimation Process

Project Size – Metrics

1. Number of functional requirements
2. Cumulative number of functional and non-functional requirements
3. Number of Customer Test Cases
4. Number of 'typical sized' use cases
5. Number of inquiries
6. Number of files accessed (external, internal, master)
7. Total number of components (subsystems, modules, procedures, routines, classes, methods)
8. Total number of interfaces
9. Number of System Integration Test Cases
10. Number of input and output parameters (summed over each interface)
11. Number of Designer Unit Test Cases
12. Number of decisions (if, case statements) summed over each routine or method
13. Lines of Code, summed over each routine or method

Table 1.1. Availability of Size Estimation Metrics

	Development Phase	Available Metrics
a	Requirements Gathering	1, 2, 3
b	Requirements Analysis	4, 5
d	High Level Design	6, 7, 8, 9
e	Detailed Design	10, 11, 12
f	Implementation	12, 13

Function Points

STEP 1: measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an unadjusted function point count (UFC). Counts are made for the following categories

– External inputs – those items provided by the user that describe distinct application-oriented data (such as file names and menu selections)
– External outputs – those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)
– External inquiries – interactive inputs requiring a response
– External files – machine-readable interfaces to other systems
– Internal files – logical master files in the system

STEP 2: Multiply each number by a weight factor, according to complexity (simple, average or complex) of the parameter, associated with that number. The value is given by a table:

Table.1.2. Parameters and Complexity

Parameter	simple	average	complex
users inputs	3	4	6
users outputs	4	5	7
users requests	3	4	6
files	7	10	15
external interfaces	5	7	10

STEP 3: Calculate the total UFP (Unadjusted Function Points)
STEP 4: Calculate the total TCF (Technical Complexity Factor) by giving a value between 0 and 5 according to the importance of the following points:

Technical Complexity Factors:
1. Data Communication
2. Distributed Data Processing
3. Performance Criteria
4. Heavily Utilized Hardware
5. High Transaction Rates
6. Online Data Entry
7. Online Updating
8. End-user Efficiency
9. Complex Computations
10. Reusability
11.Ease of Installation
12. Ease of Operation

13. Portability
14. Maintainability
STEP 5: Sum the resulting numbers too obtain DI (degree of influence)
STEP 6: TCF (Technical Complexity Factor) by given by the formula
– TCF=0.65+0.01*DI
STEP 6: Function Points are by given by the formula
– FP=UFP*TCF

COCOMO 1

Table.1.3. COCOMO 1

Project complexity	Formula	Description
Simple	$PM = 2.4 (KDSI)^{1.05} \times M$	Well-understood applications developed by small teams.
Moderate	$PM = 3.0 (KDSI)^{1.12} \times M$	More complex projects where team members may have limited experience of related systems.
Embedded	$PM = 3.6 (KDSI)^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures.

COCOMO 2

✦ COCOMO 81 was developed with the assumption that a waterfall process would be used and that all software would be developed from scratch.

▪Since its formulation, there have been many changes in software engineering practice and COCOMO 2 is designed to accommodate different approaches to software development.

▪COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.

▪The sub-models in COCOMO 2 are:

- Application composition model. Used when software is composed from existing parts.

- Early design model. Used when requirements are available but design has not yet started.

- Reuse model. Used to compute the effort of integrating reusable components. Post-architecture model. Used once the system architecture has been designed and more information about the system is available.

Application composition model

▪ Supports prototyping projects and projects where there is extensive reuse.
▪ Based on standard estimates of developer productivity in application (object) points/month.
▪ Takes CASE tool use into account.
▪ Formula is
  - PM = (NAP ´ (1 - %reuse/100)) / PROD
  - PM is the effort in person-months, NAP is the number of application points and PROD is the productivity.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-II- SOFTWARE ENGINEERING– SBS1204

SBS1204 - SOFTWARE ENGINEERING

UNIT 2

Introduction – The software requirement specifications – Formal specification techniques – Languages and processors for requirements specification : SDAT, SSA, GIST, PSL/PSA, REL/REVS- Software prototyping – rapid prototyping techniques- user interface prototyping- Analysis and modeling – data, functional and behavioral models – Structured analysis and data dictionary.

REQUIREMENTS ENGINEERING TASKS

- Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification,and managing the requirements as they are transformed into an operational system
- The Reuirement enginering process is accomplished through the execution of seven distinct functions
  Inception,Elicitation,Elaboration,Negotation,Specification,Validation and Management

Inception

- How does a software project get started?
- At project inception, you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation.

- Ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.
  - **Problems of scope**. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives
  - **Problems of understanding**. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer.
  - **Problems of volatility**. The requirements change over time.

Elaboration

- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.
- This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.

Negotiation

- It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources.
- It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."
- You have to reconcile these conflicts through a process of negotiation.
- Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction

Specification

- A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
- The specification is the final work product produced by the reuirement engineer.

Validation

- Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.
- The primary requirements validation mechanism is the formal technical review

Requirements management

- Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds
- Once requirements have been identified, traceability tables are developed Among many possible traceability tables are the following:
  Features traceability table. Shows how requirements relate to important customer observable system/product features.
  Source traceability table. Identifies the source of each requirement.
  Dependency traceability table. Indicates how requirements are related to one another.
  Subsystem traceability table. Categorizes requirements by the subsystem(s) that they govern.
  Interface traceability table. Shows how requirements relate to both internaland external system interfaces.

REQUIREMENTS ANALYSIS

- Requirements analysis is a software engineering task that bridges the gap betweensystem level requirements engineering and software design
- Requirements engineering activities result in the specification of software's operational characteristics, indicate software's interface with other system elements, and establish constraints that software must meet.
- Requirements analysis allows the software engineer (sometimes called *analyst* in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software.
- Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs.
- Finally,the requirements specification provides the developer and the customer with the means to assess quality once software is built.

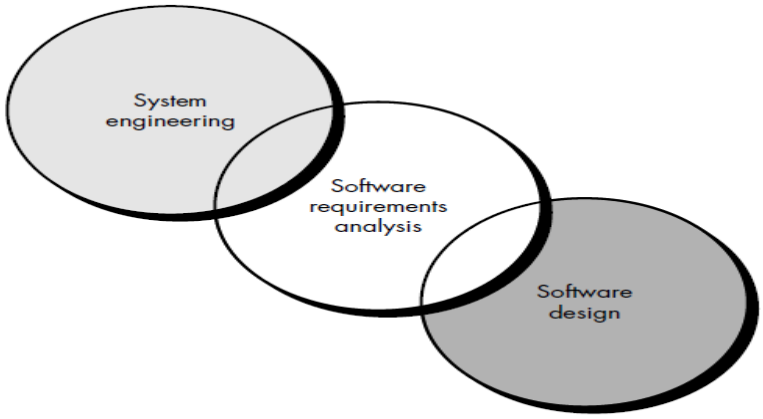


Fig.2.1. Analysis as a bridge between system engineering and software design

REQUIREMENTS ELICITATION FOR SOFTWARE

- Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process.

Initiating the Process

- The most commonly used requirements elicitation technique is to conduct a meeting or interview.
- **The analyst start by asking context-free questions. For example, the analyst might ask:**
  - Who is behind the request for this work?
  - Who will use the solution?
  - What will be the economic benefit of a successful solution?
  - Is there another source for the solution that you need?
- **The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution:**
  - How would you characterize "good" output that would be generated by a successful solution?

- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?
- **The final set of questions focuses on the effectiveness of the meeting.**
- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Facilitated Application Specification Techniques

- Facilitated application specification techniques (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements . The basic guidelines:
- A meeting is conducted at a neutral site and attended by both software engineers and customers.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

Quality Function Deployment

- Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- QFD "concentrates on maximizing customer satisfaction from the software engineering process"
- QFD identifies three types of requirements
- **Normal requirements.**

- The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied.
- Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.
- **Expected requirements.**
- These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.
- Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

- **Exciting requirements.**

- These features go beyond the customer's expectationsand prove to be very satisfying when present.
- For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product

Use-Cases

- It is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users.
- To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called **use cases** ,provide a description of how the system will be used.
- To create a use-case, the analyst must first identify the different types of people (or devices) that use the system or product. These actors actually represent roles that people (or devices) play as the system operates.
- An actor is anything that communicates with the system or product and that is external to the system itself.

The Software Requirements Specification

- The Software Requirements Specification is produced at the culmination of the analysis task.
- The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.
- The National Bureau of Standards, IEEE and the U.S. Department of Defense have all proposed candidate formats for software requirements specifications .

The **Introduction** of the software requirements specification states the goals and objectives of the software, describing it in the context of the computer-based system.

The **Information Description** provides a detailed description of the problem that the software must solve. Information content, flow, and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.

A description of each function required to solve the problem is presented in the **Functional Description**. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are

included to graphically represent the overall structure of the software and interplay among software functions and other system elements.

The **Behavioral Description** section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.

Validation Criteria is probably the most important and, ironically, the most often neglected section of the Software Requirements Specification. How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints?

Finally, the specification includes a **Bibliography and Appendix**. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

In many cases the Software Requirements Specification may be accompanied by an executable prototype, a paper prototype or a Preliminary User's Manual. The Preliminary User's Manual presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

SPECIFICATION REVIEW

A review of the Software Requirements Specification (and/or prototype) is conducted by both the software developer and the customer. Because the specification forms the foundation of the development phase, extreme care should be taken in conducting the review.

The review is first conducted at a macroscopic level; that is, reviewers attempt to ensure that the specification is complete, consistent, and accurate when the overall information, functional, and behavioral domains are considered.

Once the review is complete, the Software Requirements Specification is "signedoff" by both the customer and the developer. The specification becomes a "contract" for software development. Requests for changes in requirements after the specification is finalized will not be eliminated. But the customer should note that each after the-fact change is an extension of software scope and therefore can increase cost and/or protract the schedule.

Even with the best review procedures in place, a number of common specification problems persist. The specification is difficult to "test" in any meaningful way, and therefore inconsistency or omissions may pass unnoticed. During the review, changes to the specification may be recommended. It can be extremely difficult to assess the global impact of a change; that is, how a change in one function affects requirements for other functions.

Modern software engineering environments incorporateCASE tools that have been developed to help solve these problems.

ANALYSIS PRINCIPLES

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

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

Davis suggests a set of guiding principles for requirements engineering:

- **Understand the problem** before you begin to create the analysis model.
- **Develop prototypes that enable a user to understand how human/machine interaction will occur.**.
- **Record the origin of and the reason for every requirement.**
- **Use multiple views of requirements.** Building data, functional, and behavioral models provide the software engineer with three different views.
- **Rank requirements.** If an incremental process model is applied, those requirements to be delivered in the first increment must be identified.
- **Work to eliminate ambiguity.** Because most requirements are described in a natural language. The use of formal technical reviews is one way to uncover and eliminate ambiguity

The Information Domain

- Software is built to process data, to transform data from one form to another; that is, to accept input,manipulate it in some way, and produce output.
- Software also processes events.
- An event represents some aspect of system control and is really nothing more than Booleandata—it is either on or off, true or false, there or not there.
- For example, a pressure sensor detects that pressure exceeds a safe value and sends an alarm signal to monitoring software.
- The information domain contains three differentviews of the data and control as each is processed by a computer program:
    1. Information content and relationships
    2. Information flow
    3. Information structure

- **Information content**represents the individual data and control objects that constitute some larger collection of information transformed by the software.

- For example, the data object, **paycheck,** is a composite of a number of important pieces ofdata: the payee's name, the net amount to be paid, the gross pay, deductions, and so forth.
- Therefore, the content of **paycheck** is defined by the attributes that are needed to create it.
- **Information flow**represents the manner in which data and control change as eachmoves through a system
- **Information structure**represents the internal organization of various data and controlitems.

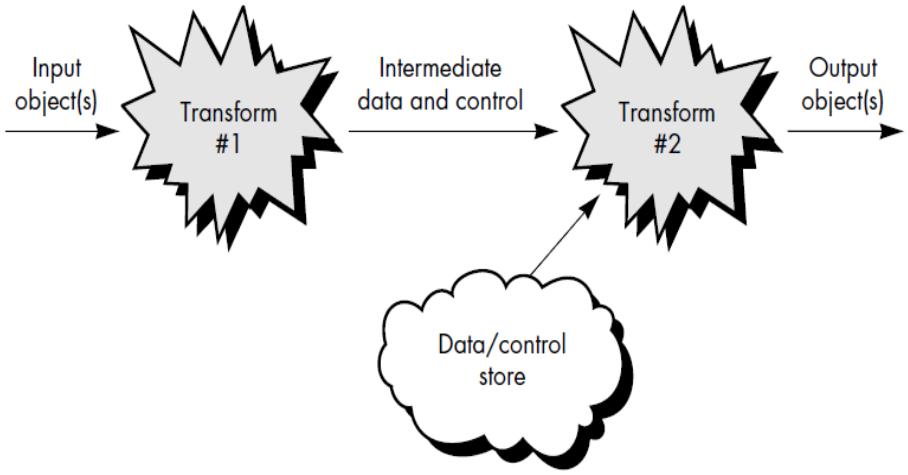


Fig.2.2. Information flow and transformation

Modeling

- We create functional models to gain a better understanding of the actual entity to be built.
- It must be capable of representing the information that software transforms, the functions that enable the transformation to occur and the behavior of the system as the transformation is taking place.

Functional models.

- Software transforms information, and in order toaccomplish this, it must perform at least three generic functions: input, processing,and output.
- The functional model begins with a single context level model. Over a series of iterations, more and more functional detail is provided, until a thorough delineation of all system functionality is represented.

Behavioral models.

- Most software responds to events from the outsideworld.
- This stimulus/response characteristic forms the basis of the behavioralmodel.
- A computer program always exists in some state—an externallyobservable mode of behavior thatis changed only when some event occurs

Models created during requirements analysis serve a number of important roles:

- The model aids the analyst in understanding the information, function, andbehavior of a system, thereby making the requirements analysis task easierand more systematic.

- The model becomes the focal point for review and, therefore, the key to adetermination of completeness, consistency, and accuracy of the specifications.
- The model becomes the foundation for design, providing the designer withan essential representation of software that can be "mapped" into an implementationcontext.

Partitioning

- Problems are often too large and complex to be understood as a whole.
- For this reason, we tend to partition such problems into parts that can be easily understood and establish interfaces between the parts so that overall function can be accomplished.
- We establish a hierarchical representation of function or information and then partition the uppermost element by
  (1) Exposing increasing detail by moving verticallyin the hierarchy or
  (2) Functionally decomposing the problem by moving horizontallyin the hierarchy.

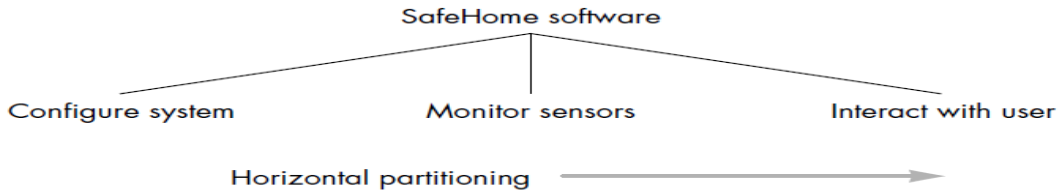


Fig.2.3. Horizontal partitioning of SafeHome function

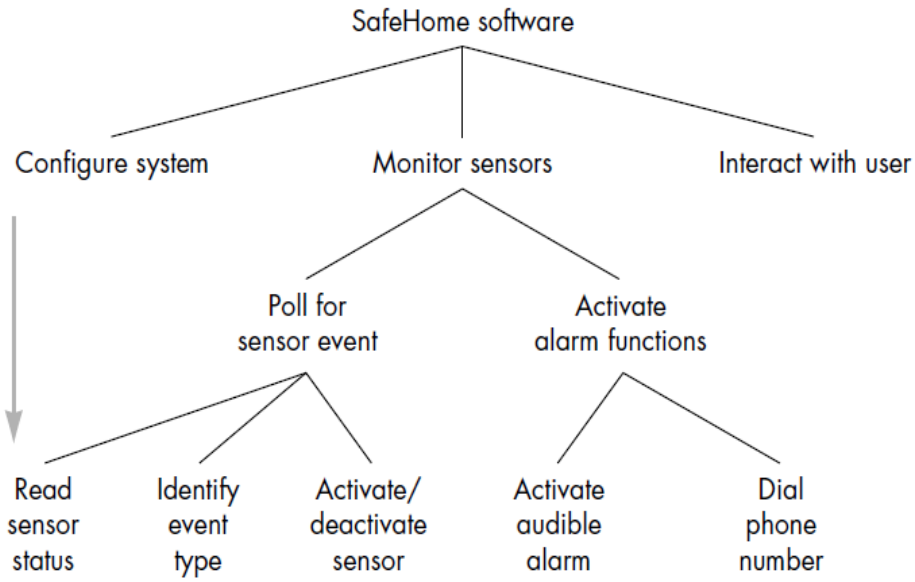


Fig.2.4. Vertical partitioning of SafeHome function

LANGUAGES AND PROCESSORS FOR REQUIREMENTS SPECIFICATION

Many other approaches have been proposed for requirements analysis and specification. These approaches help to arrange information and provide an automated analysis of requirements specification of the software. In addition, these approaches are used for organizing and specifying the requirements. The specification language used for modeling can be either graphical (depicting requirements using diagrams) or textual (depicting requirements in text form). Generally, the approaches used for analysis and specification include SADT, PSL/ PSA, RSL/REVS, SSA and GIST

Problem Statement Language (PSL) is a textual language, which is developed to describe the requirements of information systems. The **Problem Statement Analyzer (PSA)** is the processor that processes the requirements specified in PSL and then generates reports. PSL/PSA helps to document and communicate the software requirements. This approach is useful for requirements analysis as well as design. An advantage of PSA is that it allows the system to be customized according to a particular problem domain and particular solution methods because PSA is capable of defining new PSL constructs and format reports. PSL/PSA is used in commercial data processing applications, air defense systems, and so on.

PSL consists of a set of objects, where each object has properties and relationships with each other. The objective of PSL is to describe the information included in software requirements specification about the system. In PSL, this system description comprises several, namely, system input/ output flow, system structure, and data structure. **System input/output flow** describes the interaction of the system with its environment. It also provides information about the inputs received and outputs produced. **System structure** specifies the hierarchies among objects within the system. **Data structure** describes the relationships among the data used within the system and how data is manipulated by the system.

PSA operates on the information stored in the database, which is collected from the PSL description of requirements.

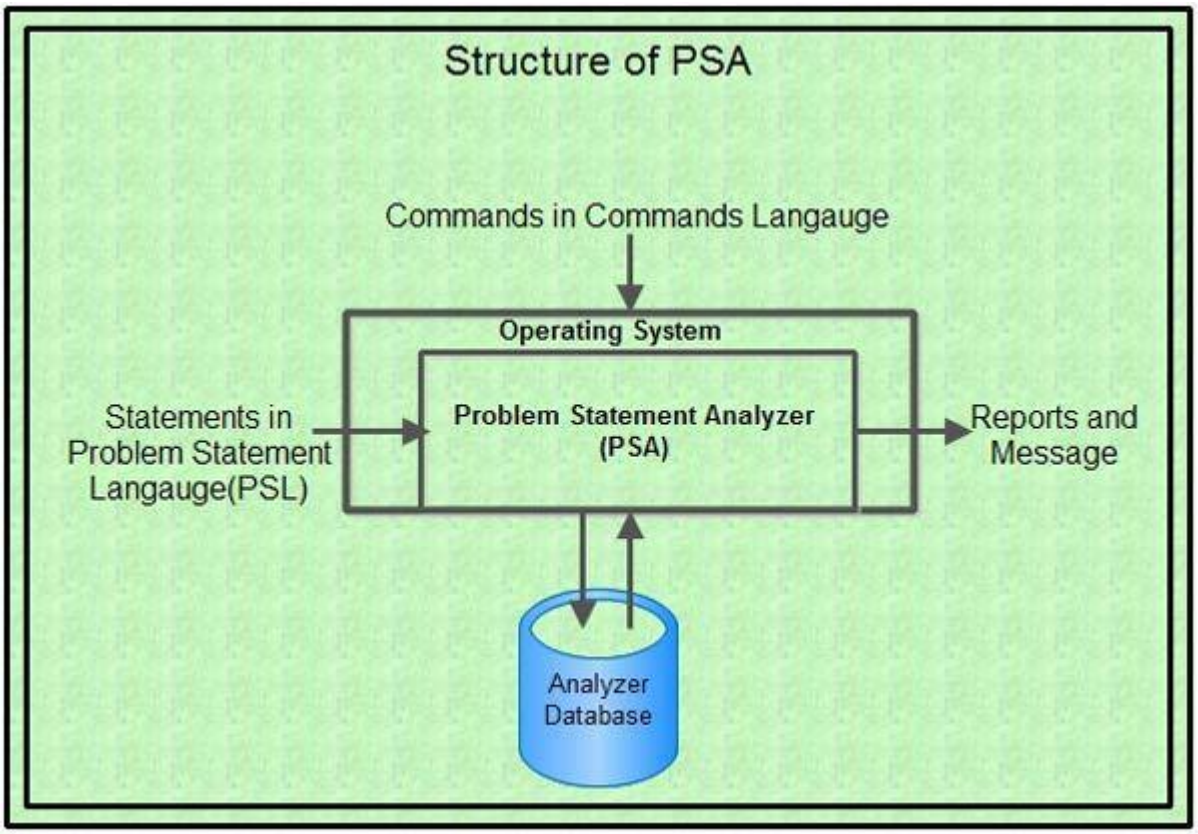


Fig.2.5. Structure of PSA

The PSA generates the following reports.

1. **Database modification report:** Specifies the changes made since the last report including the warning messages. It also provides information about the changes that have occurred due to the correction of errors.
2. **Reference report:** Includes several reports such as name list report, formatted problem statement report, and dictionary report. Name list report describes all the objects in the database. Formatted problem statement report describes the properties and relationships of a specific object. Dictionary report provides the data dictionary.
3. **Summary report:** Specifies the information gathered from various relationships. It consists of several reports such as database summary report, structure report, and external picture report. Database summary report provides information about the total number of objects used within the system including their details. Structure report represents the information in the form of hierarchy. External picture report describes the data-flow in a graphical form.
4. **Analysis report:** Includes information about inputs and outputs and problems related to inconsistency within the system. Analysis report comprises various reports such as contents comparison report, data processing interaction report, and processing chain report. Contents comparison report compares the similarity of inputs and outputs. Data processing report helps to find inconsistency in information flow and unused data objects. Processing chain report specifies the dynamic behavior of the system.

The **Requirements Statement Language (RSL)** is developed for real-time control systems. The **Requirements Validation System (REVS)** processes and analyzes the RSL statements. Note that both RSL and REVS are components of **Software Requirements Engineering**

Methodology (SREM). SREM helps to generate requirements for real-time systems as these systems perform critical tasks and hence require that the constraints applied on them be documented and tracked. Like PSL, RSL also uses basic concepts such as elements (describe objects), attributes (describe features of elements), relationships (describe relations between elements), and structures (consist of nodes and processing steps). RSL follows the flow-oriented approach to describe real-time systems. It represents the process control systems in terms of stimulus and response. Each flow in RSL starts with a stimulus and continues till the final response is achieved. When requirements are defined in such a sequence, processing steps are required. The execution of a processing step may involve various software and hardware components. REVS operates on the RSL statements. Generally, RSL comprises the following components.

1. Translator for RSL
2. Abstract system semantic model (ASSM), which is a centralized relational database and similar to PSL/PSA database
3. A set of automated tools, which is used for processing information in ASSM.

Some examples of automated tools are interactive graphics package, static checker, and automated simulation package. Interactive graphics package facilitates in describing flow paths, **static checker** checks the completeness and consistency of the information within the system, and **automated simulation package** generates and executes simulation models of the system.

Note that REVS is a large and complex software tool. Due to this, its use is cost effective only for the specification of large and complex real-time systems. However, the RSL notation can be applied manually to describe the characteristics of a real time system.

Structured Analysis and Design Technique (SADT) uses a graphical notation, and is generally applied in information processing systems. It comprises two parts, namely, Structured Analysis (SA) and Design Technique (DT). SA describes the requirements with the help of diagrams whereas DT specifies how to interpret the results.

The model of SADT consists of an organized collection of SA diagrams. These diagrams facilitate software engineers to identify the requirements in a structured manner by following a top-down approach and decomposing system activities, data, and their relationships. The text embedded in these diagrams is written in natural language, thus, specification language is a combination of both graphical language and natural language. The commonly-used SA diagrams include activity diagram (actigram) and data diagram (datagram). Both activity and data diagrams comprise nodes and arcs. Note that each diagram must consist of 3 to 6 nodes including the interconnecting arcs. These diagrams are similar to a data-flow diagram as they follow a top-down approach but differ from DFD as they may use loops, which are not used in a DFD.

An activity diagram is shown with nodes and arcs. The nodes represent the activities and the arcs describe the data-flow between the activities. Four different types of arcs can be connected to each node, namely, input data, control data, processor, and output data. **Input data** is the data that are transformed to output(s). **Control data** is the data that constrain the kind or extent of process being described. **Processor** describes the mechanism, which is in the form of tools and techniques to perform the transformation. **Output data** is the result produced after sending input, performing control activity, and mechanism in a system. The arcs on the left side of a node indicate inputs and the arcs on the right side indicate outputs. The arcs entering from the top of a node describe the control whereas the arcs entering from

the bottom describe the mechanism. The data-flows are represented with the help of inputs and outputs while the processors represent the mechanism.

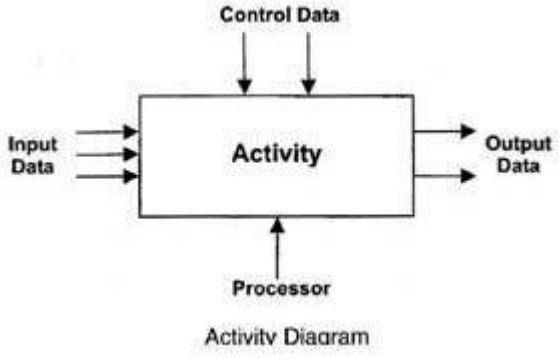


Fig.2.6. Activity Diagram

A data diagram is shown with nodes and arcs, which are similar to that of an activity diagram. The nodes describe the data objects and the arcs describe the activities. A data diagram also uses four different types of arcs. The arcs on the left side indicate inputs and the arcs on the right side indicate the output. Here, input is the activity that creates a data object whereas output is the activity that uses the data object. The 'control activity' (arcs entering from top) controls the conditions in which the node is activated and the 'storage device' (arcs entering from bottom) indicates the mechanism for storing several representations of a data object. Note that in both the diagrams, controls are provided by the external environment and by the outputs from other nodes.

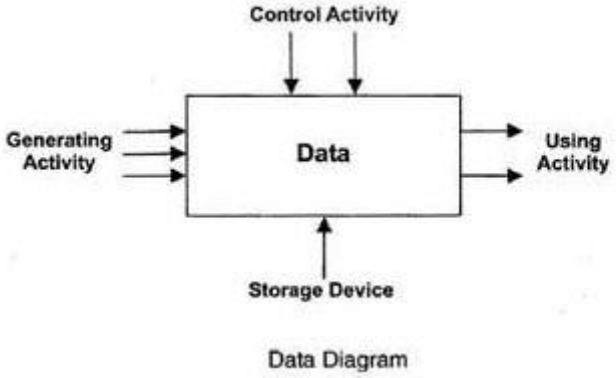


Fig.2.7. Data Diagram

Structured analysis and the design technique provide a notation and a set of techniques, which facilitate to understand and record the complex requirements clearly and concisely. The top-down approach used in SADT helps to decompose high level nodes into subordinate diagrams and to differentiate between the input, output, control, and mechanism for each node. In addition, this technique provides actigrams, datagrams, and the management techniques to develop and review an SADT model. Note that SADT can be applied to all types of systems and is not confined only to software applications.

Two similar versions of Structured System Analysis (SSA) have been described by Gane and Sarson (GAN79) and by DeMarco (DEM78). The present discussion is based on the Gane and Sarson version. The DeMarco version is similar, but does not incorporate the data-base concepts of Gane and Sarson. SSA is used primarily in traditional data processing environments. Like SADT, SSA uses a graphical language to build models of systems. Unlike SADT, SSA incorporates data-base concepts; however, SSA does not provide the variety of structural mechanisms available in SADT. There are four basic features in SSA: data flow diagrams, data dictionaries, procedure logic representations, and data store structuring techniques.

SSA data flow diagrams are similar to SADT actigrams, but they do not indicate mechanism and control, and an additional notation is used to show data stores. An SSA data flow diagram is illustrated in Figure 4.24. As discussed in connection with Figure 4.2, open-ended rectangles indicate data stores, labels on the arcs denote data items, shaded rectangles depict sources and sinks for data, and the remaining rectangles indicate processing steps.

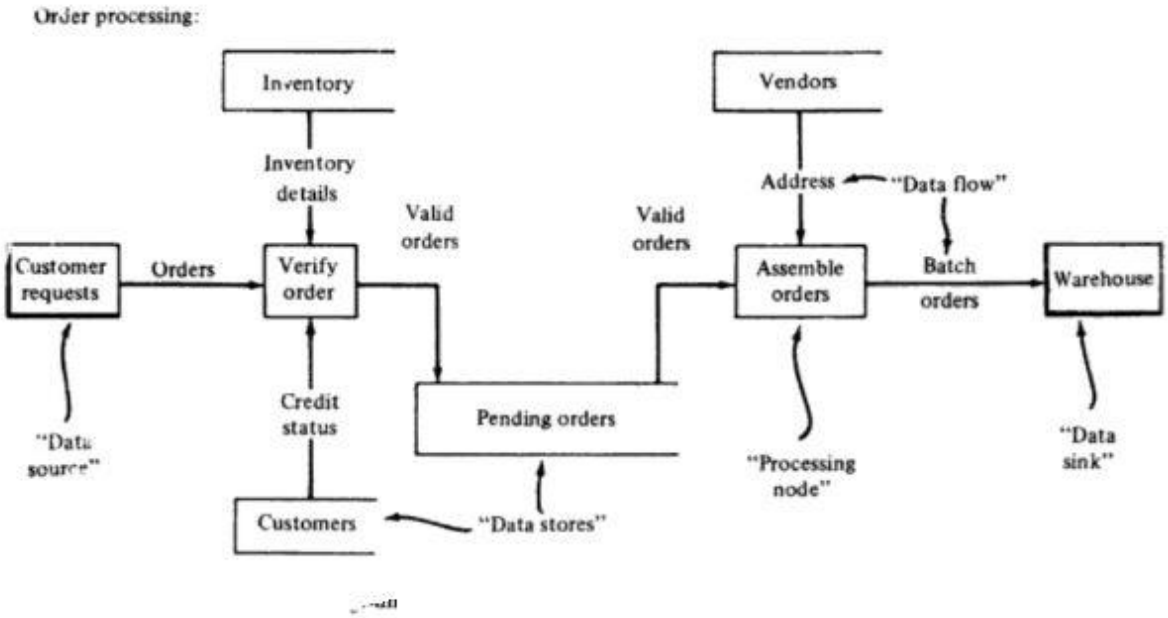


Fig.2.8. Order processing

A data dictionary is used to define and record data elements, and processing logic representations, such as decision tables and structured English, are used to specify algorithmic processing details. Processing logic representations are used to precisely specify processing sequences in terms that are understandable to customers and developers. A typical data dictionary entry is presented in Figure

DATA FLOW: ORDER

COMPOSITION:

- CUSTOMER_IDENTITY
- ORDER_DATE
- ITEM_ORDERED +
  - CATALOG_NUMBER
  - ITEM_NAME
  - UNIT PRICE
  - QUANTITY
  - TOTAL_COST

DATA STORE: CUSTOMER_IDENTITY

COMPOSITION:

- NAME
  - FIRST
  - MIDDLE
  - LAST
- PHONE
  - AREA_CODE
  - NUMBER
  - EXTENSION (Optional)
- SHIP_TO_ADDRESS
  - STREET
  - CITY
  - STATE-ZIP
- BILL_TO_ADDRESS (Same as above if empty)
  - STREET
  - CITY
  - STATE-ZIP

DATA STORE: ORDER_DATE

COMPOSITION:

- TIME
- DAY
- MONTH
- YEAR

Gist

Gist is a formal specification language developed at the USC/Information Sciences Institute by R. Balzar and colleagues (BAL81). Gist is a textual language based on a relational model of objects and attributes. A Gist specification is a formal description of valid behaviors of a system. A specification is composed of three parts:

1. A specification of object types and relationships between these types. This determines a set of possible states.
2. A specification of actions and demons which define transitions between possible states.
3. A specification of constraints on states and state transitions.

The valid behaviors of a system are those transition sequences that do not violate any constraints.

SOFTWARE PROTOTYPING

- Rapid software development to validate requirements
- Prototyping is the process of quickly putting together a working model in order to test various aspects of a design, illustrate ideas or features and gather early user feedback

Uses of prototypes

- The principal use is to help customers and developers understand the requirements for the system
- Prototyping can be considered as a risk reduction activity which reduces requirements risks

Prototyping benefits

- Misunderstandings between software users and developers are exposed
- Missing services may be detected and confusing services may be identified
- A working system is available early in the process
- The prototype may serve as a basis for deriving a system specification
- The system can support user training and system testing

Prototyping process

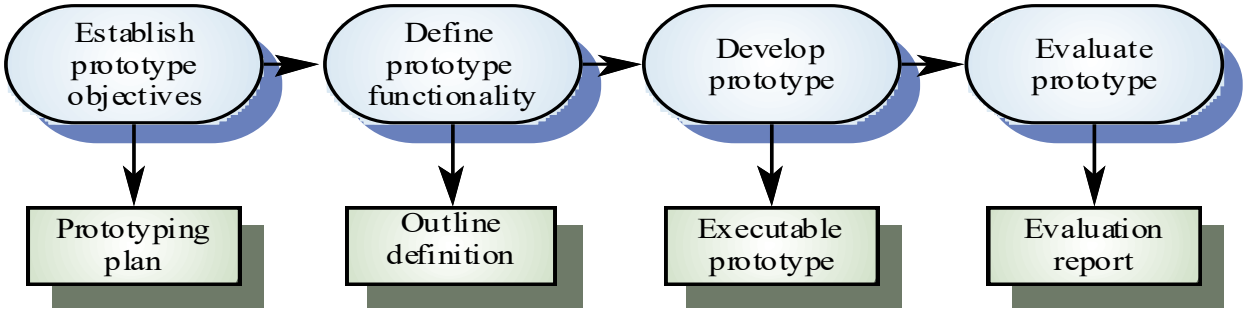


Fig.2.9. Prototyping process

Types of prototyping

Evolutionary prototyping

- An open-ended approach, calledevolutionary prototyping, uses the prototype as the first part of an analysis activity thatwill be continued into design and construction.
- The prototype of the software is thefirst evolution of the finished system.

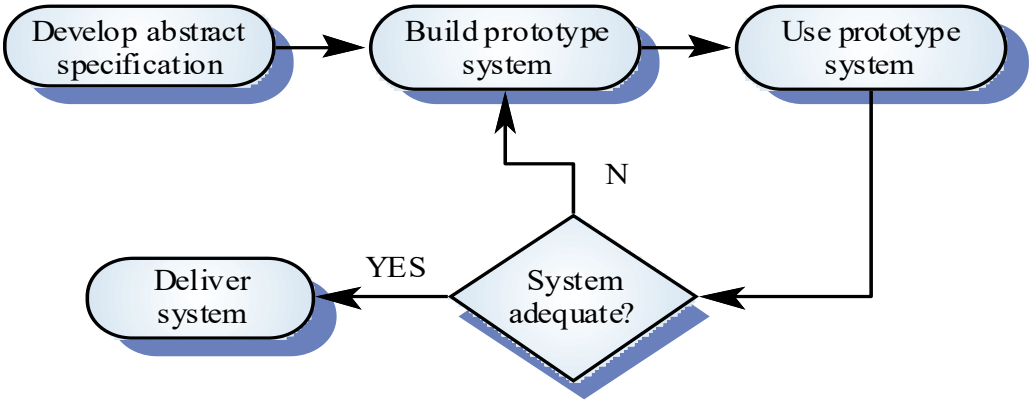


Fig.2.10. Evolutionary prototyping

Throw-away prototyping

- The close-endedapproach is often called throwaway prototyping.
- Using this approach, a prototypeserves solely as a rough demonstration of requirements. It is then discarded, and thesoftware is engineered using a different paradigm.

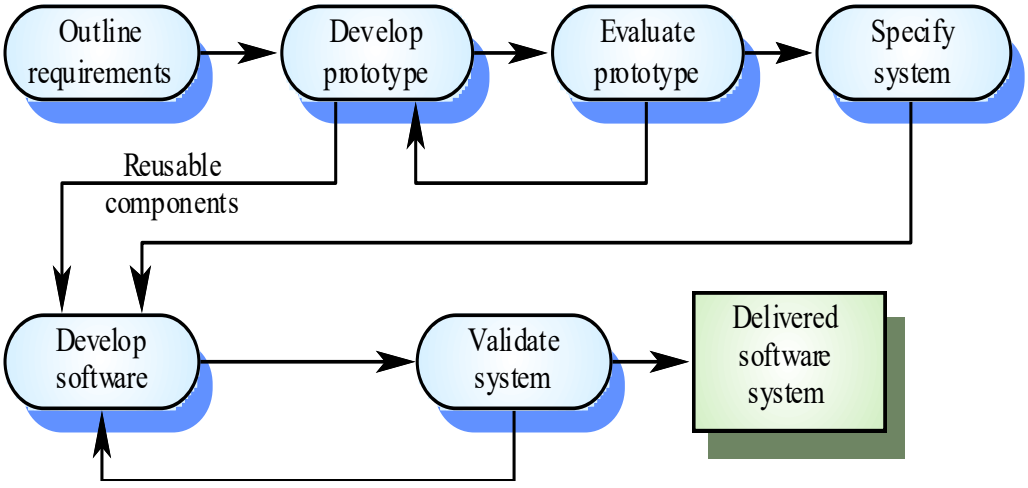


Fig.2.11. Throw-away prototyping

Table.2.1. Selecting the appropriate prototyping approach

Question	Throwaway prototype	Evolutionary prototype	Additional preliminary work required
Is the application domain understood?	Yes	Yes	No
Can the problem be modeled?	Yes	Yes	No
Is the customer certain of basic system requirements?	Yes/No	Yes/No	No
Are requirements established and stable?	No	Yes	Yes
Are any requirements ambiguous?	Yes	No	Yes
Are there contradictions in the requirements?	Yes	No	Yes

Prototyping Methods and Tools

To conduct rapid prototyping,three generic classes of methods and tools areavailable:

Fourth generation techniques.

- Fourth generation techniques (4GT)encompass a broad array of database query and reporting languages, programand application generators, and other very high-level nonprocedurallanguages.
- Because 4GT enable the software engineer to generate executablecode quickly, they are ideal for rapid prototyping.

Reusable software components.

- Another approach to rapid prototypingis to assemble, rather than build, the prototype by using a set of existing softwarecomponents.
- It should be noted that an existing softwareproduct can be used as a prototype for a "new, improved" competitiveproduct.

Formal specification and prototyping environments.

- Developers of these formal languages are in the process of developinginteractive environments that

1. Enable an analyst to interactively createlanguage-based specifications of a system or software,
2. Invoke automated tools that translate the language-based specifications into executable code,and
3. Enable the customer to use the prototype executable code to refine formal requirements.

USER INTERFACE PROTOTYPING

- It is impossible to pre-specify the look and feel of a user interface in an effective way. prototyping is essential
- UI development consumes an increasing part of overall system development costs

- User interface generators may be used to ‘draw’ the interface and simulate its functionality with components associated with interface entities
- Web interfaces may be prototyped using a web site editor

Techniques

1. **Work with the real users**. The best people to get involved in prototyping are the ones who will actually use the application when it is done. These are the people who have the most to gain from a successful implementation; these are the people who know their own needs best.
2. **Get your stakeholders to work with the prototype**. Just as if you want to take a car for a test drive before you buy it, your users should be able to take an application for a test drive before it is developed. Furthermore, by working with the prototype hands-on, they can quickly determine whether the system will meet their needs. A good approach is to ask them to work through some use case scenarios using the prototype as if it were the real system.
3. **Understand the underlying business**. You need to understand the underlying business before you can develop a prototype that will support it. The more you know about the business, the more likely it is you can build a prototype that supports it. Once again, active stakeholder participation is critical to your success.
4. **You should only prototype features that you can actually build**. If you cannot possibly deliver the functionality, do not prototype it.
5. **You cannot make everything simple**. Sometimes your software will be difficult to use because the problem it addresses is inherently difficult. Your goal is to make your user interface as easy as possible to use, not simplistic.
6. **It's about what you need**. Their point is a good user interface fulfills the needs of the people who work with it. It isn’t loaded with a lot of interesting, but unnecessary, features.
7. **Get an interface expert to help you design it**. User interface experts understand how to develop easy-to-use interfaces, whereas you probably do not. A generalizing specialist with solid UI skills would very likely be an ideal member of your development team.
8. **Explain what a prototype is**. The biggest complaint developers have about UI prototyping is their users say “That’s great. Install it this afternoon.” This happens because users do not realize more work is left to do on the system. The reason this happens is simple: From your user's point-of-view, a fully functional application is a bunch of screens and reports tied together by a menu.
9. **Consistency is critical**. Inconsistent user interfaces lead to less usable software, more programming, and greater support and training costs.
10. **Avoid implementation decisions as long as possible**. Be careful about how you name these user interface items in your requirements documents. Strive to keep the names generic, so you do not imply too much about the implementation technology.
11. **Small details can make or break your user interface**. Have you ever used some software, and then discarded it for the product of a competitor because you didn’t like the way it prints, saves files, or some other feature you simply found too annoying to use? I have. Although the rest of the software may have been great, that vendor lost my business because a portion of its product’s user interface was deficient.

ANALYSIS MODELING

Goals of Analysis Modeling

- Provides the first technical representation of a system
- Is easy to understand and maintain
- Deals with the problem of size by partitioning the system
- Uses graphics whenever possible
- Differentiates between essential information versus implementation information
- Helps in the tracking and evaluation of interfaces
- Provides tools other than narrative text to describe software logic and policy

- **Elements of the Analysis Model**

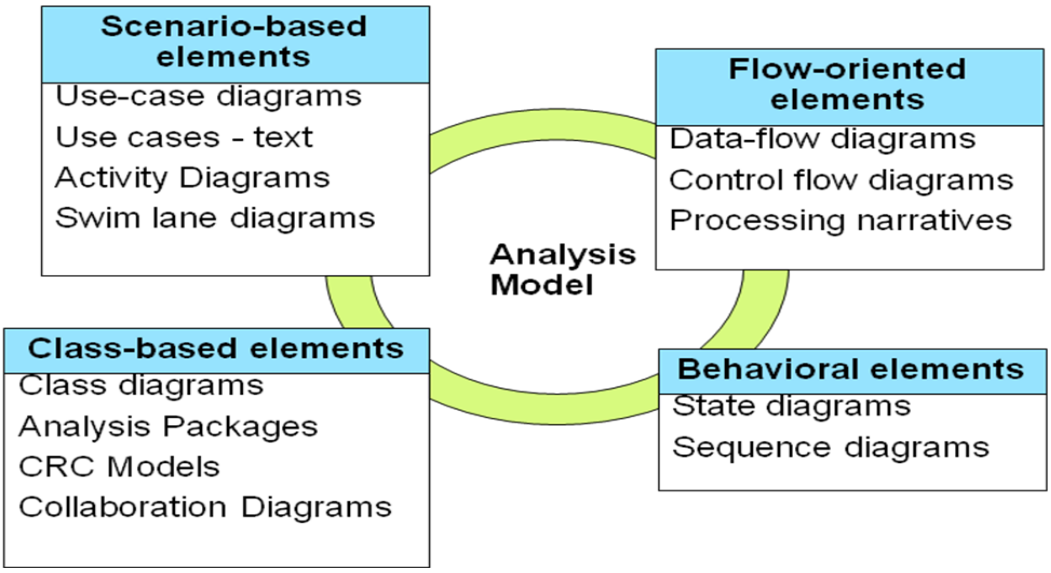


Fig.2.12. Elements of the Analysis Model

Data Modeling

- Examines data objects independently of processing
- Focuses attention on the data domain
- Creates a model at the customer's level of abstraction
- Indicates how data objects relate to one another

Data modeling concepts

Data Objects

- A data object can be an external entity , a thing ,an occurrence or event ,a role, an organizational unit, a place, or a structure.

Data Attributes

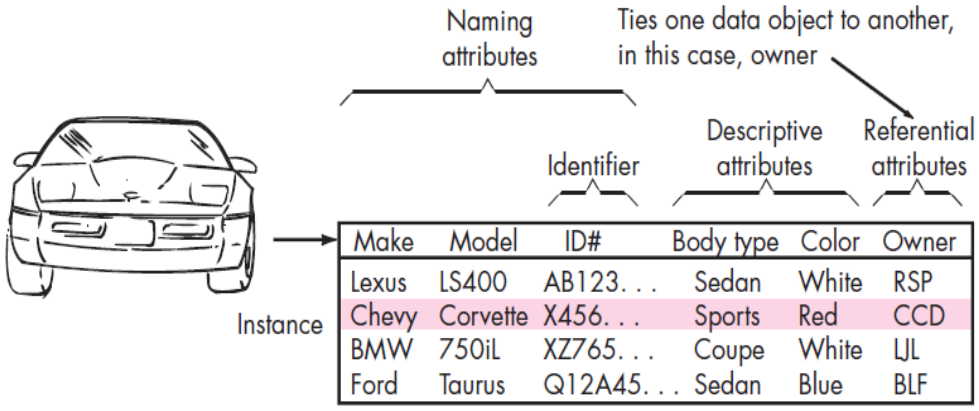
- Data attributes define the properties of a data object
- They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

(3) make reference to another instance in another table

- In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object.



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Fig.2.13. Tabularrepresentationof data objects

- **Relationships**

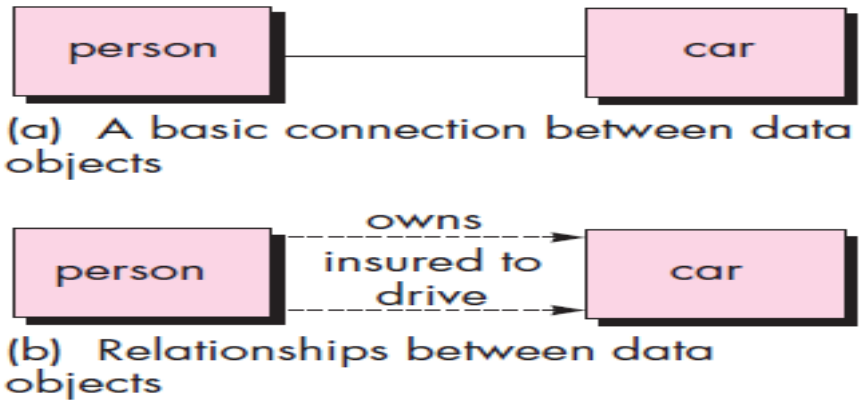


Fig.2.14. Relationships

- Data objects are connected to one another in different ways.
- Consider the two objects person and car ,a connection is established between them because they are related.
- For example
  - A person owns a car
  - A person is insured to drive a car

Cardinality and Modality

- We have defined a set of objects and represented the object/relationship pairs thatbind them. But a simple pair that states: **object X** relates to **object Y** does not provideenough information for software engineering purposes. We must understandhow many occurrences of **object X** are related to how many occurrences of **objectY.** This leads to a data modeling concept called cardinality.

Cardinality.

- The data model must be capable of representing the number of occurrencesobjects in a given relationship. The cardinality of anobject/relationship pair in the following manner:
- Cardinality is the specification of the number of occurrences of one [object] that can berelated to the number of occurrences of another [object].
- Cardinality is usually expressedas simply 'one' or 'many.Taking into consideration all combinationsof 'one' and 'many,' two [objects] can be related as
  - **One-to-one (l:l)**—An occurrence of [object] 'A' can relate to one and only one occurrenceof [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
  - **One-to-many (l:N)**—One occurrence of [object] 'A' can relate to one or many occurrencesof [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.'
  - **Many-to-many (M:N)**—An occurrence of [object] 'A' can relate to one or more occurrencesof 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.'
- Cardinality defines "the maximum number of objects that can participate in a relationship"It does not, however, provide an indication of whether or not a particulardata object must participate in the relationship.

Modality.

- The modality of a relationship is 0 if there is no explicit need for the relationshipto occur or the relationship is optional. The modality is 1 if an occurrence ofthe relationship is mandatory.

Scenario based Modeling

- If the software engineer understand how end users want to interact with a system,the software team will be able to properly charectrize requirements and build meaningful analysis and design models.
- Analysis modeling with UML begins with the creation of scenarios in the form of uses-cases,activity diagram and swimlane diagrams.
- **<u>Use-case Diagram</u>**
- In general, use cases are written first in an informal narrative fashion
- Use case: Access camera surveillance via the Internet—display camera views
- (ACS-DCV)
- Actor: homeowner
- The homeowner logs onto the SafeHome Products website.
- The homeowner enters his or her user ID.
- The homeowner enters two passwords (each at least eight characters in length).
- The system displays all major function buttons.
- The homeowner selects the "surveillance" from the major function buttons.
- The homeowner selects "pick a camera."
- The system displays the floor plan of the house.
- The homeowner selects a camera icon from the floor plan.
- The homeowner selects the "view" button.
- The system displays a viewing window that is identified by the camera ID.
- The system displays video output within the viewing window at one frame per second.

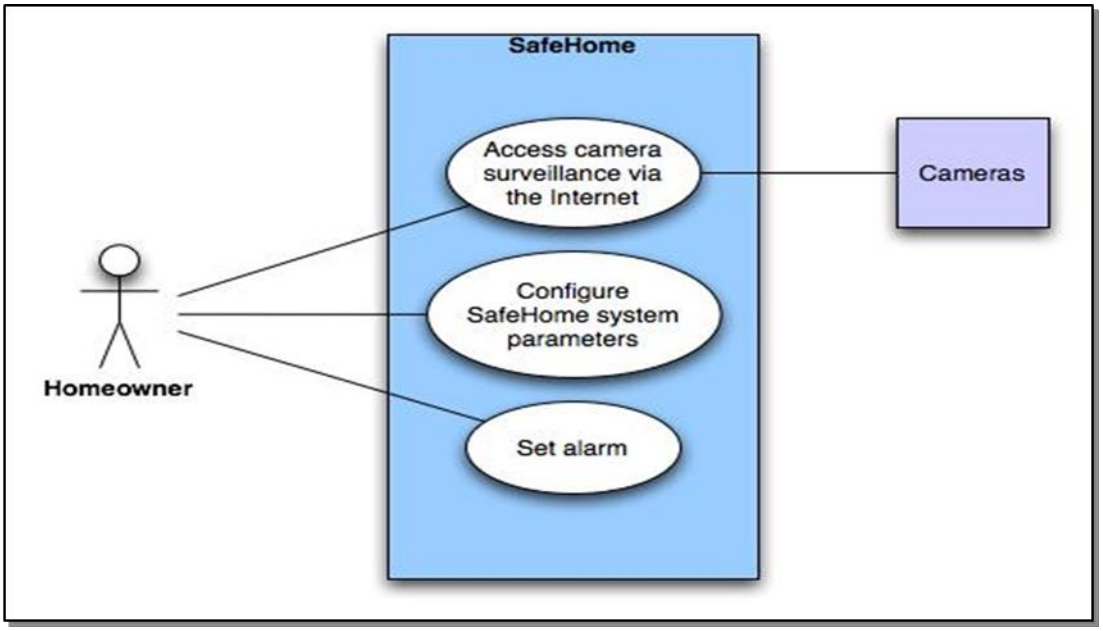


Fig.2.15. Preliminary use-case diagram for the safehome software

Alternative Actions

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error condition at this point?
- Is it possible that the actor will encounter behavior invoked by some event outside the actor's control?

Activity diagram

- The UML activity diagram supplements the use-case by providing a graphical representation of the flow of interaction within a specific scenario
- Similar to flowchart an activity diagram uses rounded rectangles to imply a specify system functions,arrows to represent flow through the system,decision diamonds to depict a branching decision and solid horizontal lines to indicate that parallel activities are occurring

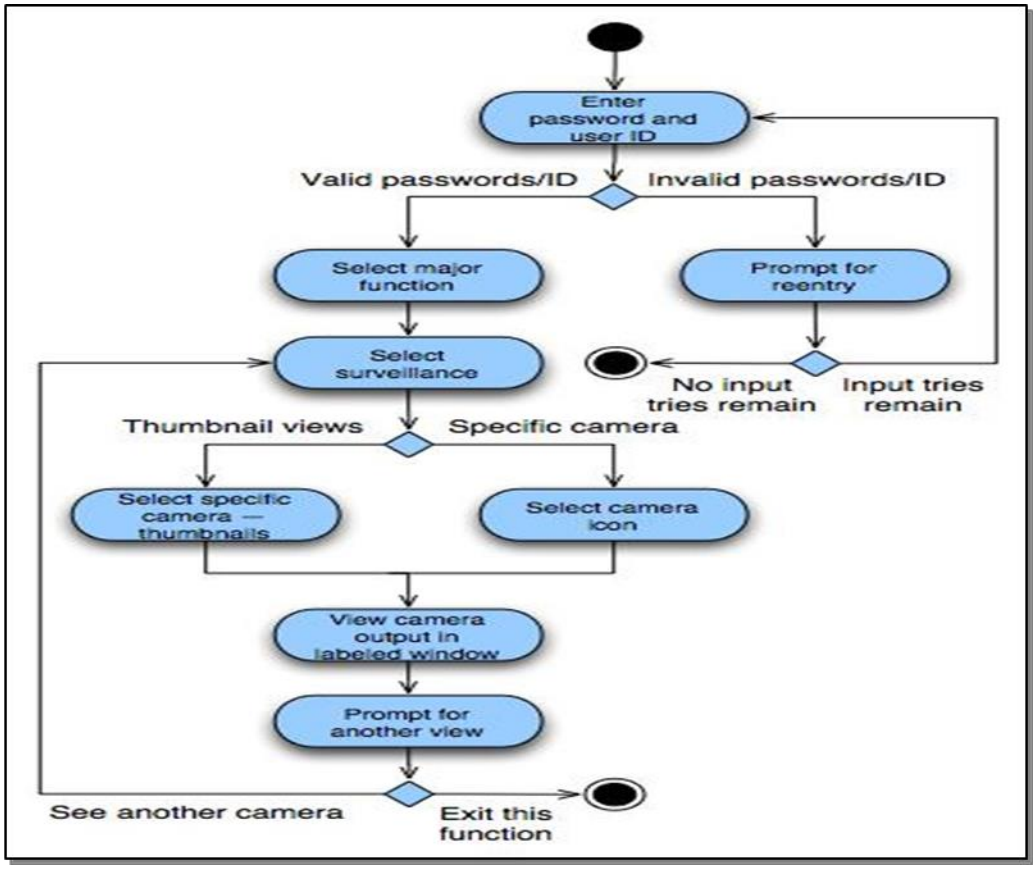


Fig.2.16. Activity diagram for access camera surveillance-display camera view functions

Swimlane diagram

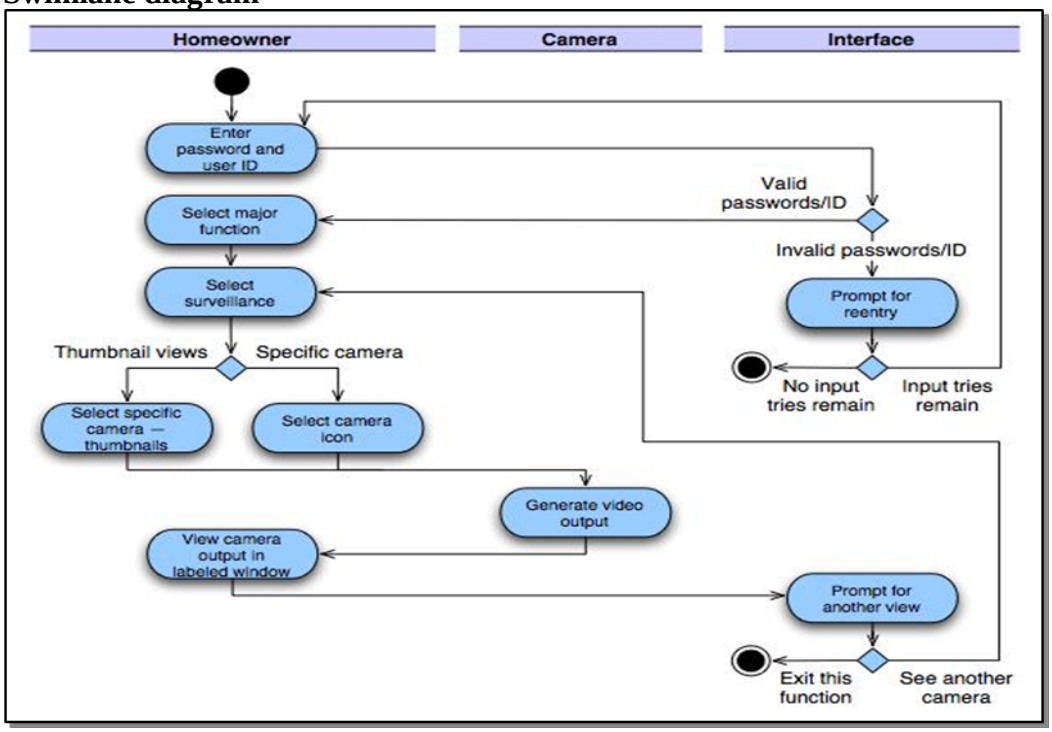


Fig.2.17. Swimlane diagram for Access camera surveillance-diplay camera views function

- The UML Swimlane diagram is a useful variation of the activity diagram and allows the modeler to represent the flow of activites described by the use-case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.
- Responsiblities are represented as parallel segments that divide the diagram vertically ,like the lanes in a swimming pool.
-

Flow-Oriented Modeling

Data Flow Model

- The DFD takes an input-process-output view of a system. That is data objects flow into the software ,transformed by processing elements,and resultant data objects flow out of the software

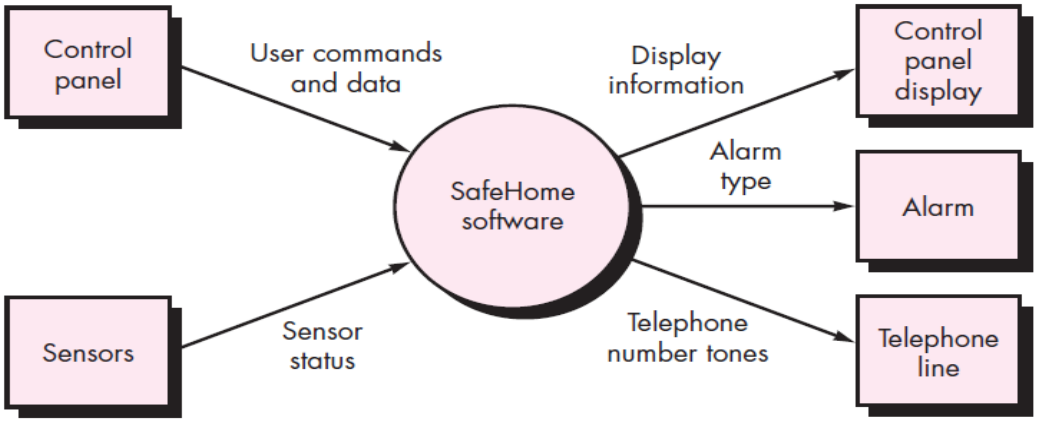


Fig.2.18.Context level DFD for the SafeHome security function

Guideliness

- The level 0 data flow diagram should epict the software /system as a single bubble.
- Primary input and output should be carefully noted.
- Refinement should begin by isolating candidate processes ,data objects and data stores to be represented at the next level.
- All arrows and bubbles should be labelled with meaningful names
- Information flow through continuity must be maintained from level to level and
- One bubble at a time should be refined.

Flow Modeling Notation

Fig.2.19. Flow modeling notation

- The level 0 DFD is now expanded into a level 1 data flow model.
- An effective approach is to perform a “grammatical parse” on the narrative that describes the context level bubble. That is we isolate all nouns and verbs in a safehome processing narrative derived during the first requirement gathering meeting.
- verbs are safehome processes,that is they may represented as bubbles in a subsequent DFD.
- Nouns are either external entities (boxes),data or control objects(arrows) or data stores(double lines).

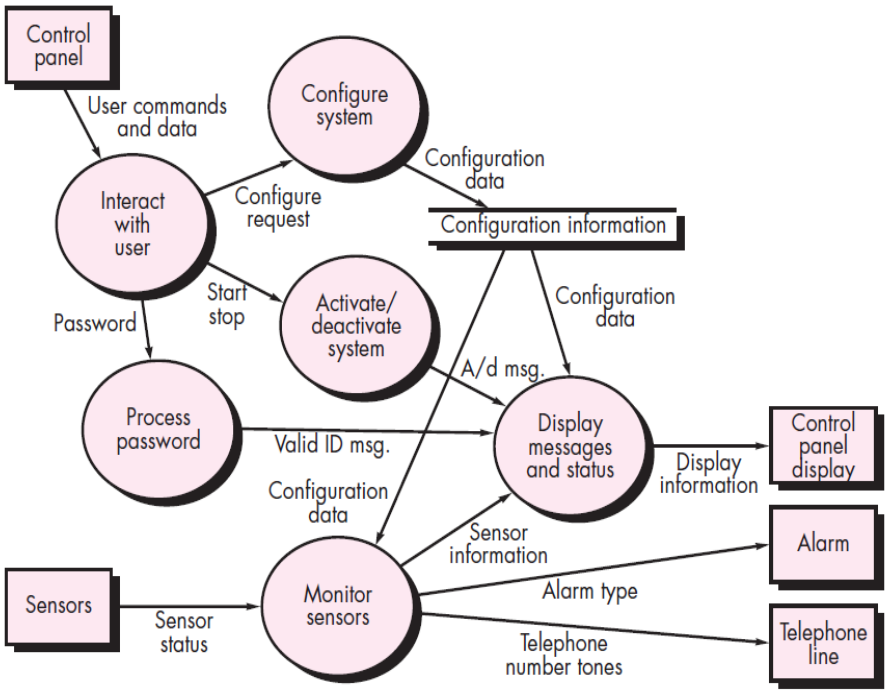


Fig.2.20. Level 1 DFD for Safe Home security function

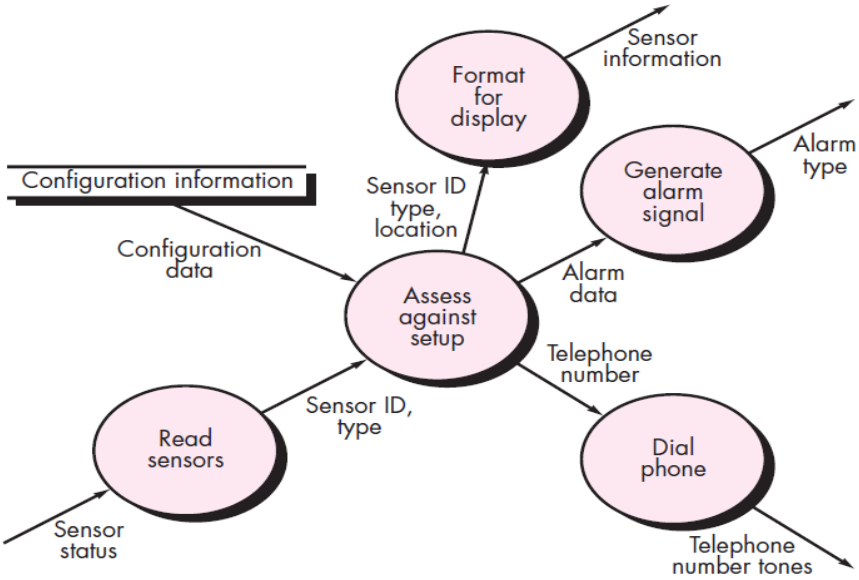


Fig.2.21. Level 2 DFD that refines the monitor sensors process

Contol Flow Model

- Large class of applications are driven by events rather than data,produce control information rather than reports or displays and process information with heavy concern for time and performance .
- Such application require the use of control flow modeling in addition to the data flow modeling .

- To select potential candidate events, the following guidelines are suggested:
  - List all sensors that are "read" by the software.
  - List all interrupt conditions.
  - List all "switches" that are actuated by an operator.
  - List all data conditions.
  - Recalling the noun/verb parse that was applied to the processing narrative,review all "control items" as possible control specification inputs/outputs.
  - Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states.
  - Focus on possible omissions—a very common error in specifying control; for example, ask: "Is there any other way I can get to this state or exit from it?"

The Control Specification

- A control specification (CSPEC) represents the behavior of the system in two different ways.
- The CSPEC contains a state diagram that is a sequential specification of behavior.
- It can also contain a program activation table—a combinatorial specification of behavior.

Behavioral modeling

- The behavioral model indicates how software will respond to external events or stimuli.
- **To create the model, you should perform the following steps:**
  - Evaluate all use cases to fully understand the sequence of interaction within the system.
  - Identify events that drive the interaction sequence and understand how these events relate to specific objects.
  - Create a sequence for each use case.
  - Build a state diagram for the system.
  - Review the behavioral model to verify accuracy and consistency.

Identifying Events with the Use Case

- In general, an event occurs whenever the system and an actor exchange information.
- "homeowner uses the keypad to key in a four-digit password." In the context of the requirements model, the object, **Homeowner,** transmits an event to the object **ControlPanel.** The event might be called password entered.
- The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model.

- It is important to note that some events have an explicit impact on the flow of control of the use case, while others have no direct impact on the flow of control.
- For example, the event password entered does not explicitly change the flow of control of the use case, but the results of the event password compared (derived from the interaction "password is compared with the valid password stored in the system") will have an explicit impact on the information and control flow of the SafeHome software.

State Representations

- In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.
- The state of a class takes on both passive and active characteristics.
- A passive state is simply the current status of all of an object's attributes. For example,the passive state of the class **Player** (in the video game application discussed in would include the current position and orientation attributes of **Player** as well as other features of **Player** that are relevant to the game (e.g., an attribute that indicates magic wishes remaining).
- The active state of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The class **Player** might have the following active states: moving, at rest, injured, being cured; trapped, lost, and so forth.
- An event (sometimes called a trigger) must occur to force an object to make a transition from one active state to another.

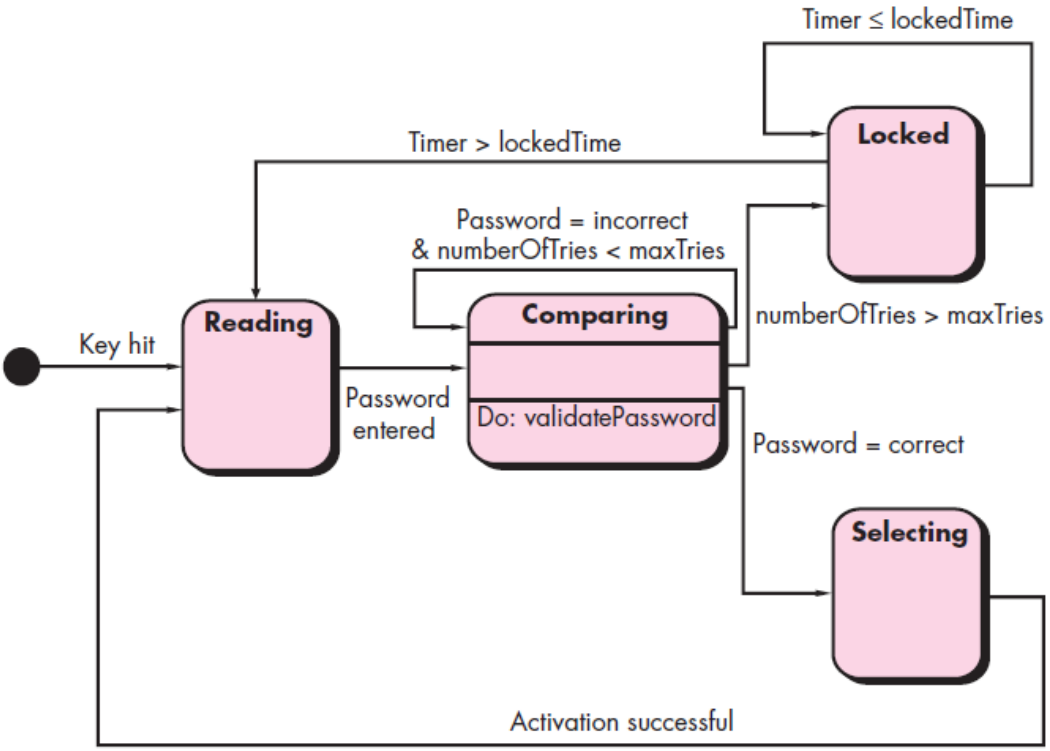


Fig.2.22. State diagrams for analysis classes

Sequence diagrams.

- The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object.
- Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time.
- In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

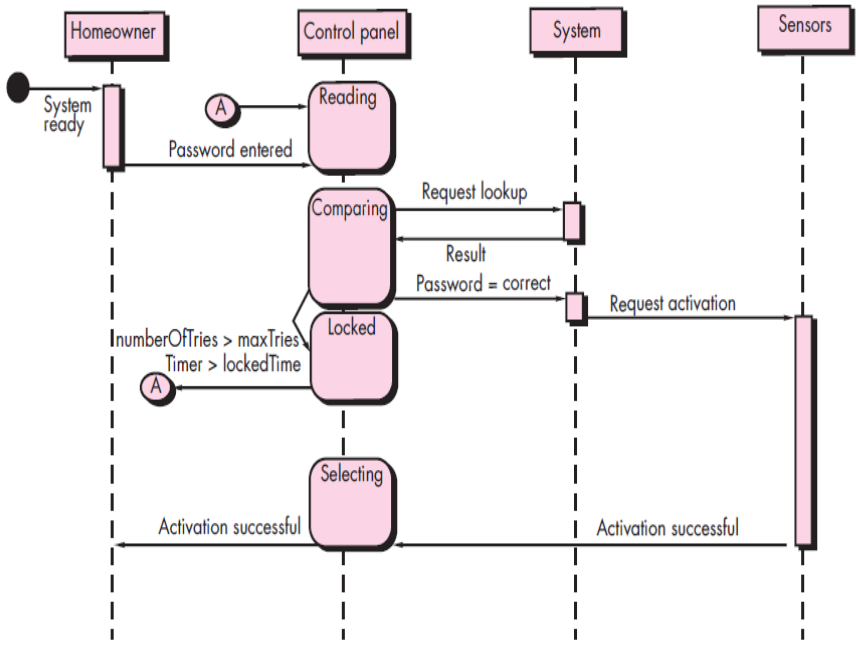


Fig.2.23. Sequence diagram

Class based modeling

- Class-based modeling represents the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined.
- The elements of a class-based model include classes and objects, attributes, operations, classresponsibility- collaborator (CRC) models, collaboration diagrams, and packages

Identifying Analysis Classes

- **External entities** (e.g., other systems, devices, people)
- **Things** (e.g., reports, displays, letters, signals)
- **Occurrences or events** (e.g., a property transfer or the completion of a series of robot movements)
- **Roles** (e.g., manager, engineer, salesperson)
- **Organizational units** (e.g., division, group, team)
- **Places** (e.g., manufacturing floor or loading dock)

- **Structures** (e.g., sensors, four-wheeled vehicles, or computers

Table 2.2. Potential classes and general classification

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

Specifying Attributes

- **System class defined for SafeHome. A homeowner** can configure the security function to reflect sensor information, alarm response information, activation/deactivation information, identification information.

identification information = system ID + verification phone number + system status

alarm response information = delay time + telephone number

activation/deactivation information = master password + number of allowable tries + temporary password

Defining Operations

- Operations define the behavior of an object

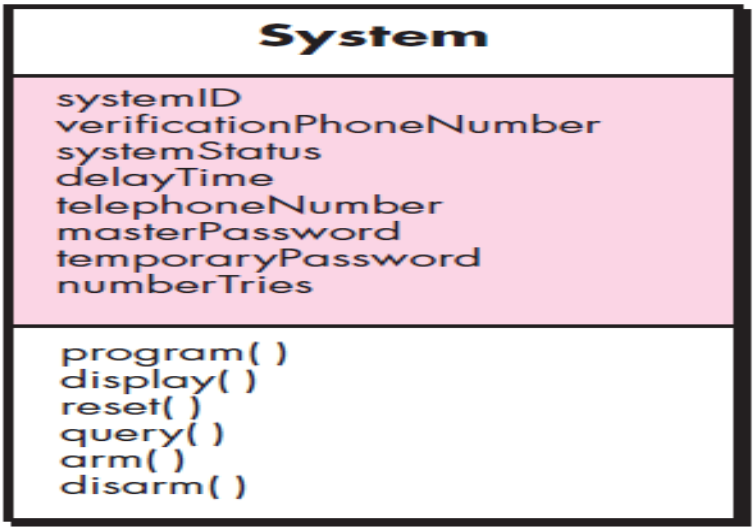
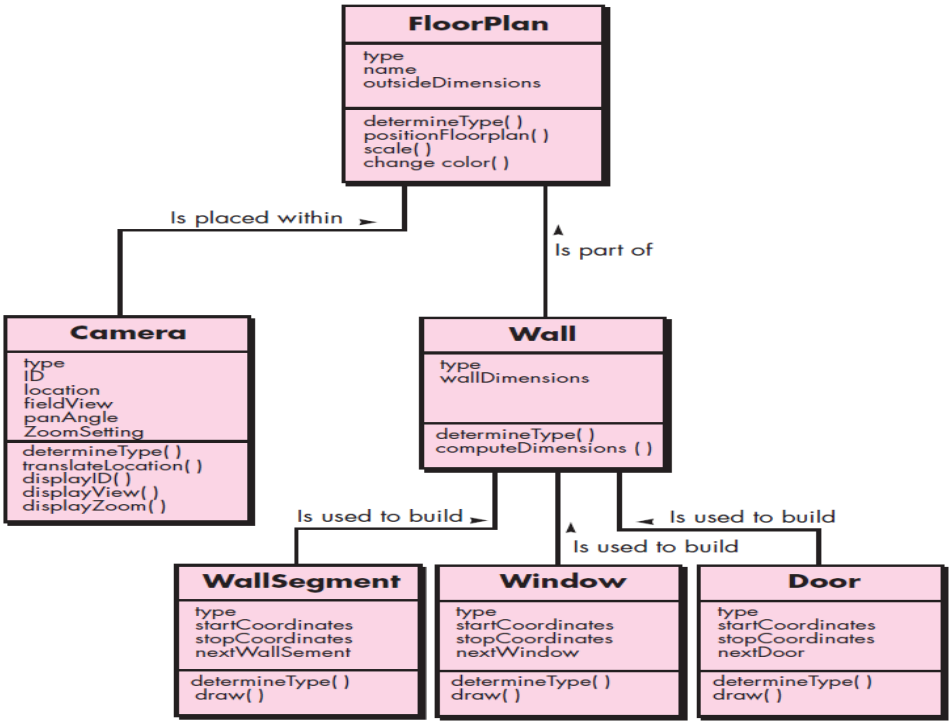


Fig.2.24. Class representation

Class-Responsibility-Collaborator (CRC) Modeling

- A CRC model is really a collection of standard index cards that represent classes.



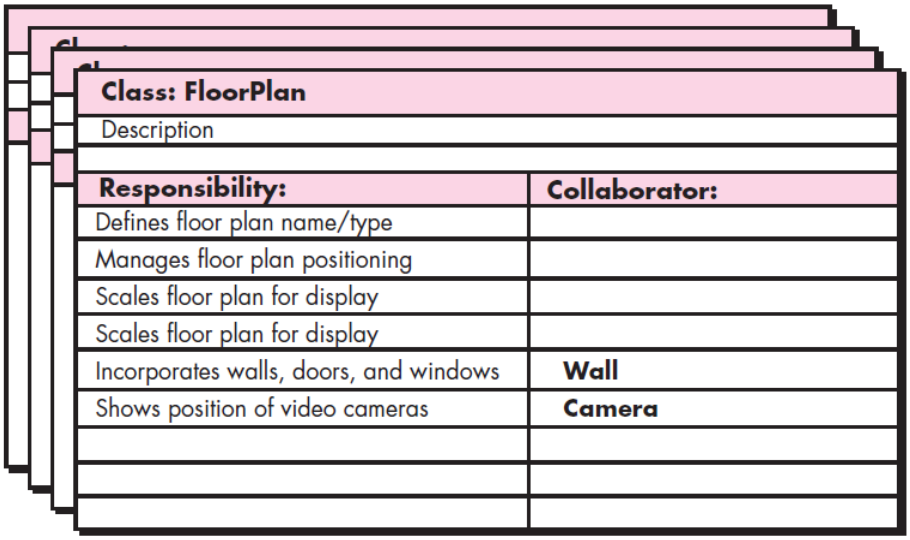


Fig.2.26. CRC model index card for floor plan

- **Collaborations.**
- A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular Responsibility
- A class can collaborate with other classes.

Data Dictionary

The data dictionary is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations. the data dictionary is always implemented as part of a CASE "structured analysis and design tool." Although the format of dictionaries varies from tool to tool, most contain the following information:

· Name—the primary name of the data or control item, the data store or an external entity.

· Alias—other names used for the first entry.

· Where-used/how-used—a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.

Data Construct	Notation	Meaning
	=	is composed of
Sequence	+	and
Selection	[|]	either-or
Repetition	{ }*n*	*n* repetitions of
	( )	optional data
	* ... *	delimits comments

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-III- SOFTWARE ENGINEERING– SBS1204

SBS1204 - SOFTWARE ENGINEERING

UNIT 3

Abstraction – Modularity – Software architecture – Cohesion, coupling – Various design concepts and notations – Real time and distributed system – Design – Documentation – Data flow oriented design – Jackson system development – Design for reuse – Programming standards. User interface Design- principles- SCM- Need for SCM- Version control – Introduction to SCM process – software configuration items

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.

Design and Software Quality

The quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs McGlaughlin three characteristics that serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Design Principles

- Design process should not suffer from "tunnel vision"
- The design should be traceable to the analysis model
- The design should not reinvent the wheel; Time is short
- The design should "minimize intellectual distance" between the software and the problem in the real world
- The design should exhibit uniformity and integration
- The design should be structured to accommodate change
- The design should be structured to degrade gently.
- Design is not coding, coding is not design
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual errors

Design Concepts

Fundamental concepts which provide foundation to design correctly:

• Abstraction

• Refinement

• Modularity

• Software Architecture

• Control Hierarchy

• Structural Partitioning

• Data Structure

• Software Procedure

• Information Hiding

Abstraction

• Identifying important features for representation

• There are many levels of abstraction depending on how detailed the representation is required

• Data abstraction - representation of data objects

• Procedural abstraction - representation of instructions

Refinement

• Stepwise refinement - top-down design strategy by Niklaus Wirth

• Refinement is actually a process of elaboration

• Starting at the highest level of abstraction, every step of refinement „decompose" instructions into more detailed instructions

• Complementary to abstraction

Modularity

Software is divided into separately named and addressable components, often called modules, that are integrated to satisfy problem requirements.

• "Divide and conquer" approach - problem is broken into manageable pieces

• Solutions for the separate pieces then integrated into the whole system

Divide and Conquer

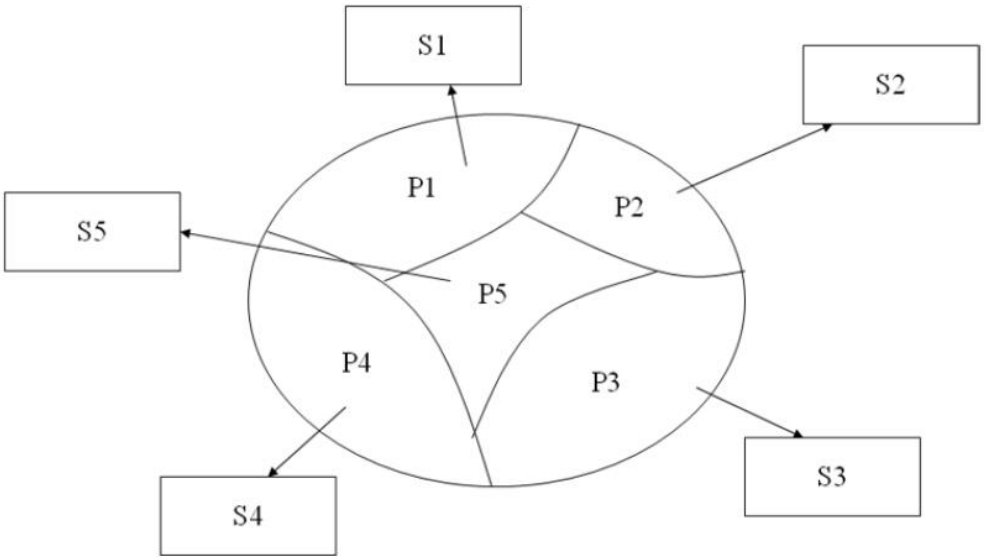


Fig.3.1. Divide and Conquer

Software Architecture

• Modules can be integrated in many ways to produce the system

• Software architecture is the overall structure of the software

• The hierarchy of components and how they interact, and the structure of data used by the components

• Use of framework models, and possible reuse of architectural patterns

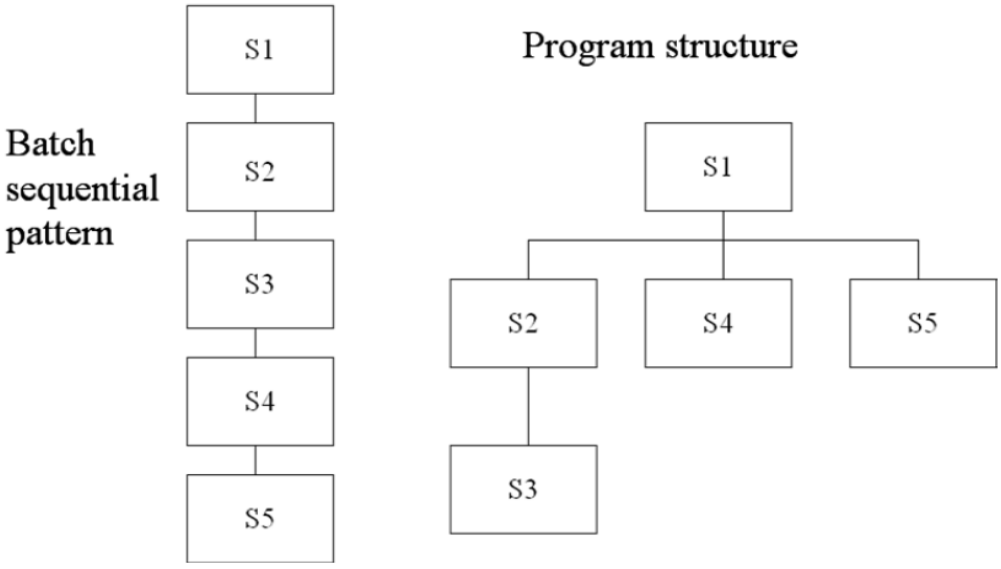


Fig.3.2. Software Architecture

Control Hierarchy

• Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control.

• Hierarchy of modules representing the control relationships

• A super-ordinate module controls another module

• A subordinate module is controlled by another module

• Measures relevant to control hierarchy: depth, width, fan-in, fan-out

• Depth and width provide an indication of the number of levels of control and overall span of control.

• Fan-out is a measure of the number of modules that are directly controlled by anothermodule. Fan-in indicates how many modules directly control a given module.

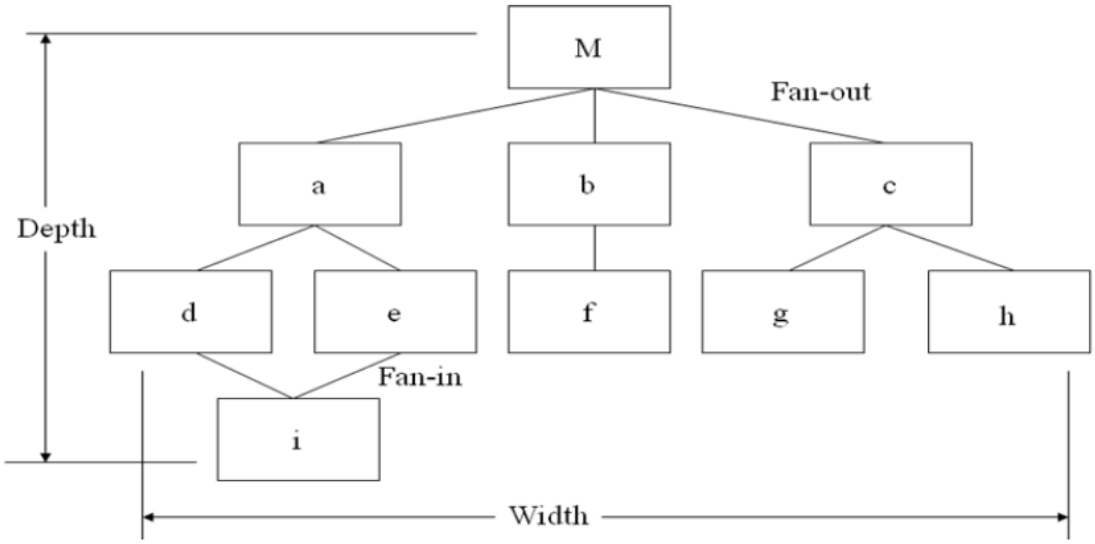


Fig.3.3. Control Hierarchy

Structural Partitioning

• Program structure partitioned horizontally and vertically

• Horizontal partitioning defines separate branches for each major program function - input, process, output

• Vertical partitioning defines control (decision-making) at the top and work at the bottom

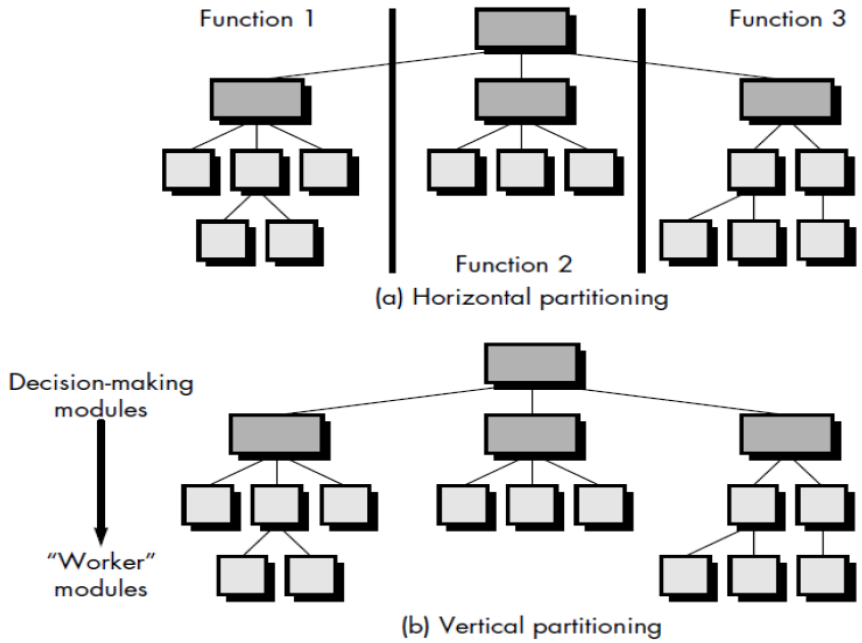


Fig.3.3. Structural Partitioning

Software Procedure

• Processing details of individual modules

• Precise specification of processing, including sequence of events, exact decision points, repetitive operations, and data organization/structure

• Procedure is layered - subordinate modules must be referenced in processing details

Information Hiding

• Information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information

• Effective modularity is achieved by independent modules, that communicate only necessary information

• Ease of maintenance - testing, modification localized and less likely to propagate

Data Structure

• Data structure is a representation of the logical relationship among individual elements of data.

Modular Design

• Functional Independence

• Designing modules in such a way that each module has specific functional requirements.

Functional independence is measured using two terms cohesion and coupling.

Cohesion

 Internal interaction of the module.

 Cohesion is a measure of relative functional strength of a module

 The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component.

Types of cohesion

- Logical Cohesion
- Coincidental cohesion
- Temporal Cohesion
- Procedure Cohesion
- Communication Cohesion
- Sequential cohesion
- Informational cohesion
- Functional cohesion
- Coincidental cohesion

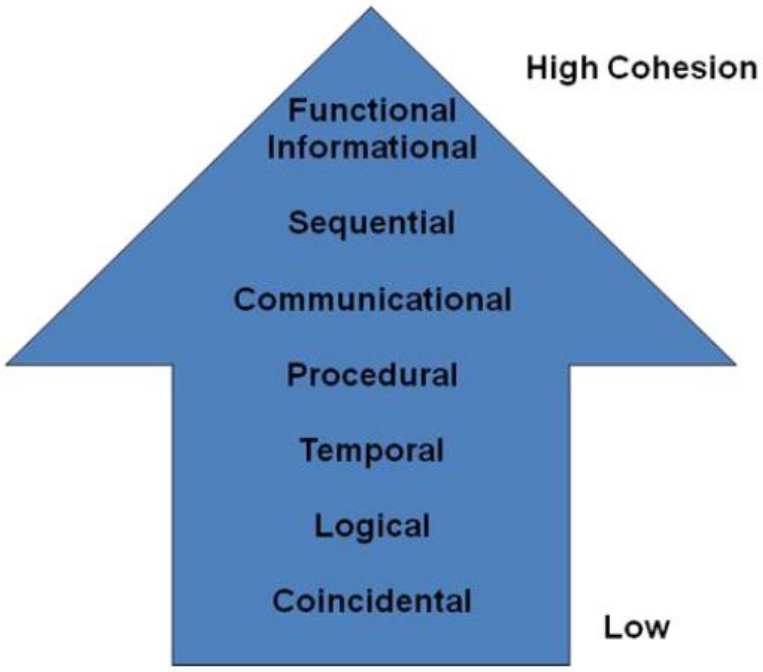


Fig.3.4. Range of Cohesion types

Coincidental cohesion

- Parts of a component are simply bundled together
- The result of randomly breaking the project into modules to gain the benefits of having multiple smaller files/modules to work on
- Usually worse than no modularization
- A module has coincidental cohesion if it performs multiple, completely unrelated actions

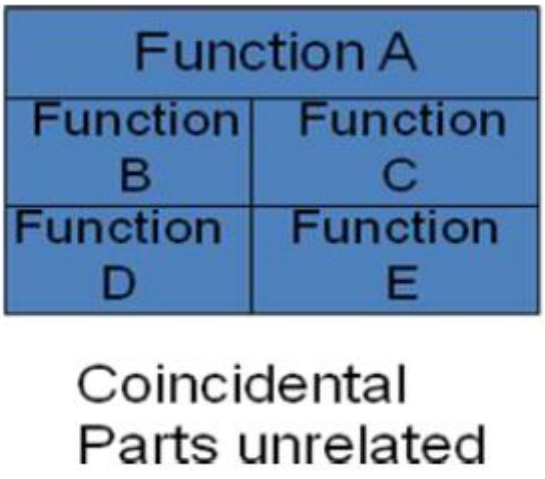


Fig.3.5. Coincidental cohesion

Logical Cohesion

- Elements of component are related logically and not functionally
- Results in hard to understand modules with complicated logic

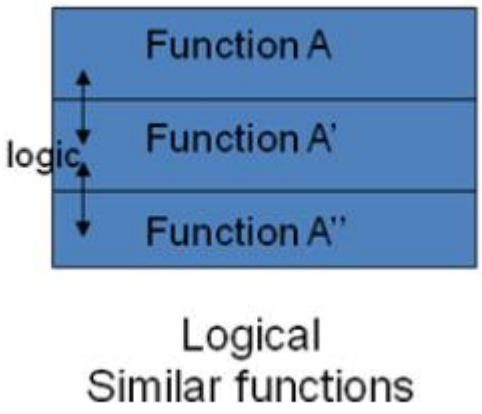


Fig.3.6. Logical cohesion

Temporal Cohesion

- Elements of a component are related by timing

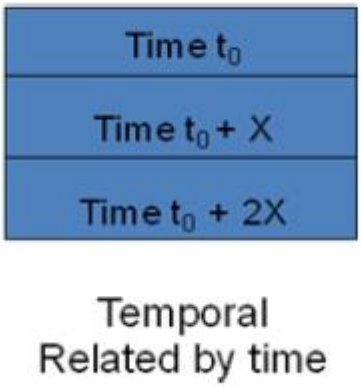


Fig.3.7. Temporal cohesion

Procedural Cohesion

Elements of a component are related only to ensure a particular order of execution.

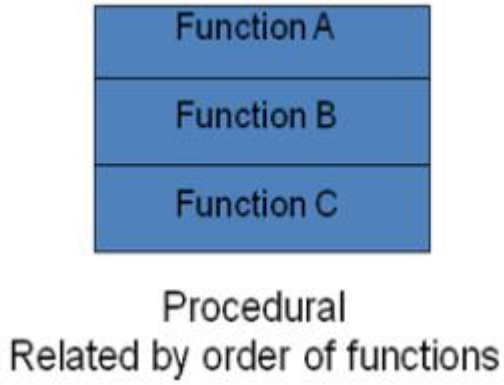


Fig.3.8. Procedural Cohesion

Communicational Cohesion

- Module performs a series of actions related by a sequence of steps to be followed by the product and all actions are performed on the same data

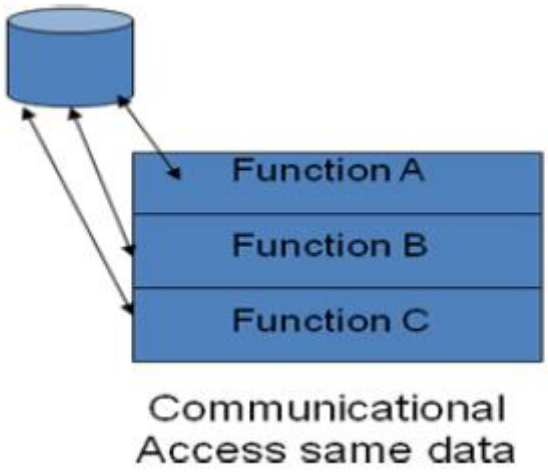


Fig.3.9. Communicational Cohesion

Sequential Cohesion

- The output of one component is the input to another.
- Occurs naturally in functional programming languages

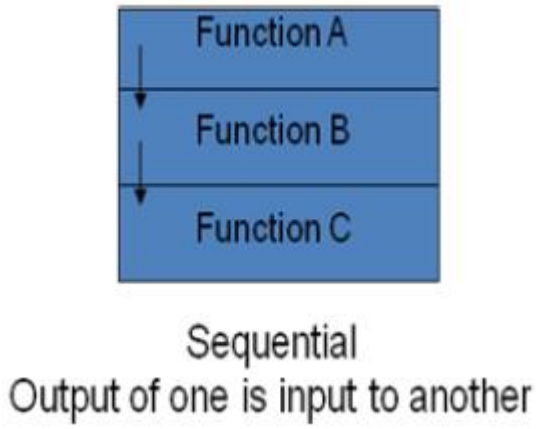


Fig.3.10. Sequential Cohesion

Informational Cohesion

- Module performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data.

Functional Cohesion

- Module with functional cohesion focuses on exactly one goal or "function"
- Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation

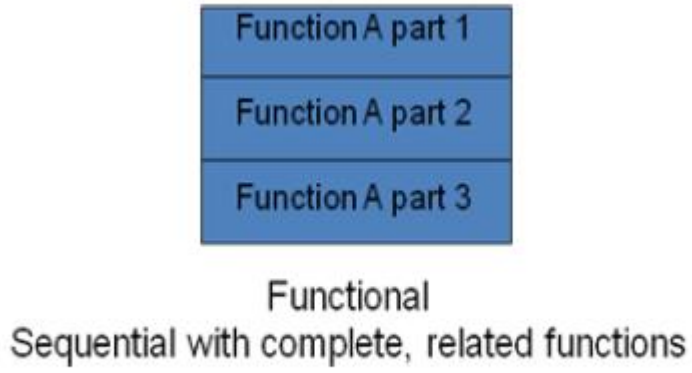


Fig.3.11. Functional Cohesion

Coupling

- **Coupling** is a measure of relative independence among modules, that is it is a measure of interconnection among modules.

- **Loose coupling** means component changes are unlikely to affect other components.
- Shared variables or control information exchange lead to **tight coupling**.
- Loose coupling can be achieved by state decentralization (as in objects) and component communication via parameters or message passing.

Tight Coupling

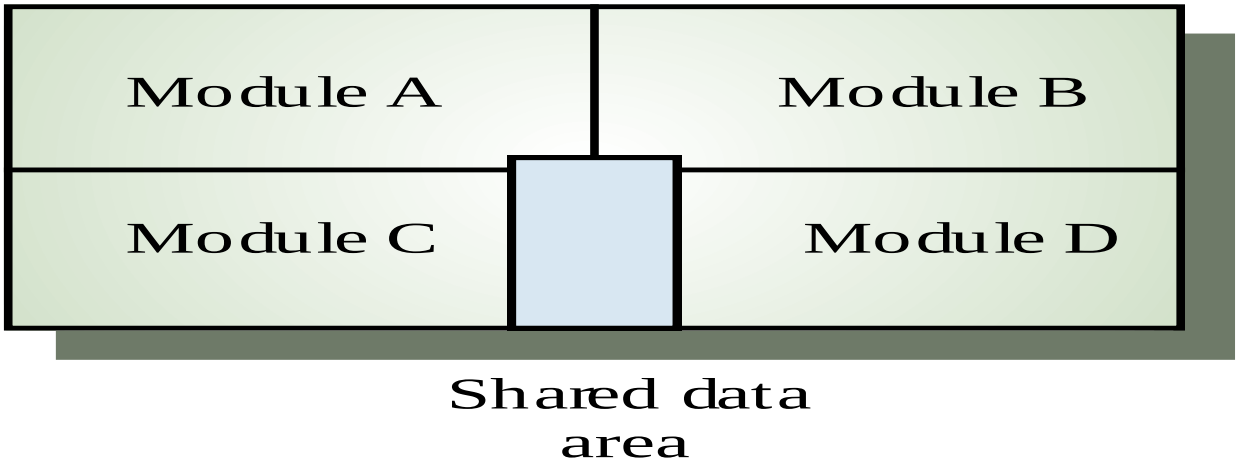


Fig.3.12. Tight Coupling

Loose Coupling

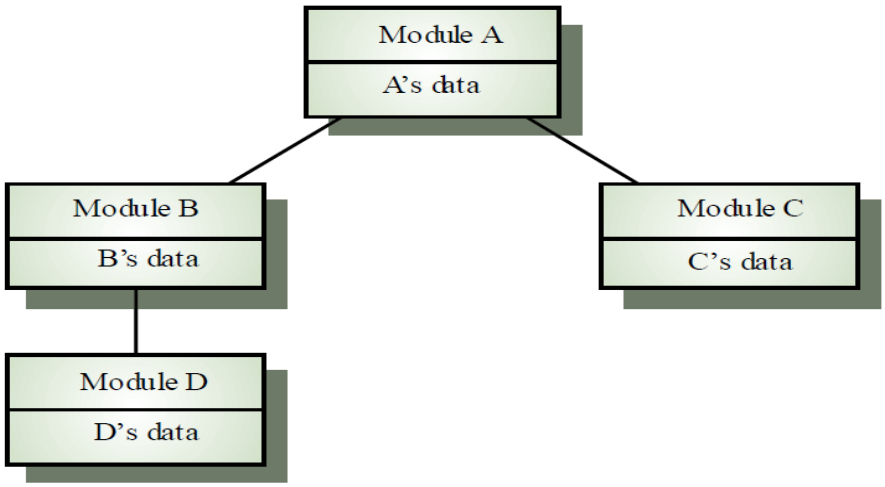


Fig.3.13. Loose Coupling

Types of coupling

- **Content coupling**
- **Common Coupling**
- **Control Coupling**
- **Stamp Coupling**
- **Data Coupling**

Content coupling

- One module directly refers to the content of the other
  - Module a modifies statement of module b

Common Coupling

- Common coupling exists when two or more modules have read and write access to the same global data.

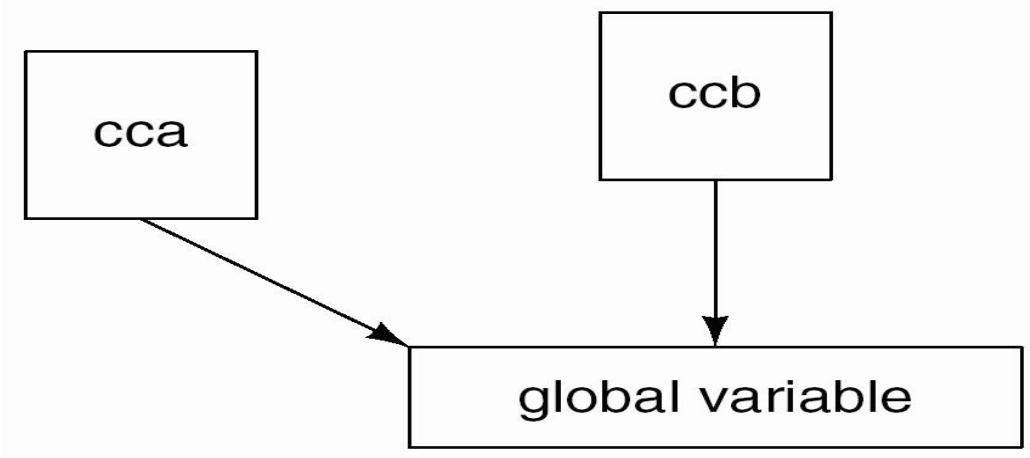


Fig.3.14. Common Coupling

Control Coupling

- Two modules are control-coupled if module 1 can directly affect the execution of module 2

Stamp Coupling

- It is a case of passing more than the required data values into a module
- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure

Data Coupling

- Two modules are data coupled if all parameters are homogeneous data items [simple parameters, or data structures all of whose elements are used by called module]

SOFTWARE ARCHITECTURE

- Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution
- The architecture of a software system defines that system in terms of computational components and interactions among those components.

ARCHITECTURAL STYLES

- An architectural style is a description of component and connector types and a pattern of their runtime control and/or data transfer.
- An architectural style, sometimes called an architectural pattern, is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems.
- Defines ways of selecting and presenting architectural building blocks

Benefits

❖ Design Reuse

❖ Code Reuse (may be domain dependant)

- ❖ Communication among colleagues
- ❖ Interoperability
- ❖ System Analysis

Data-centered architectures

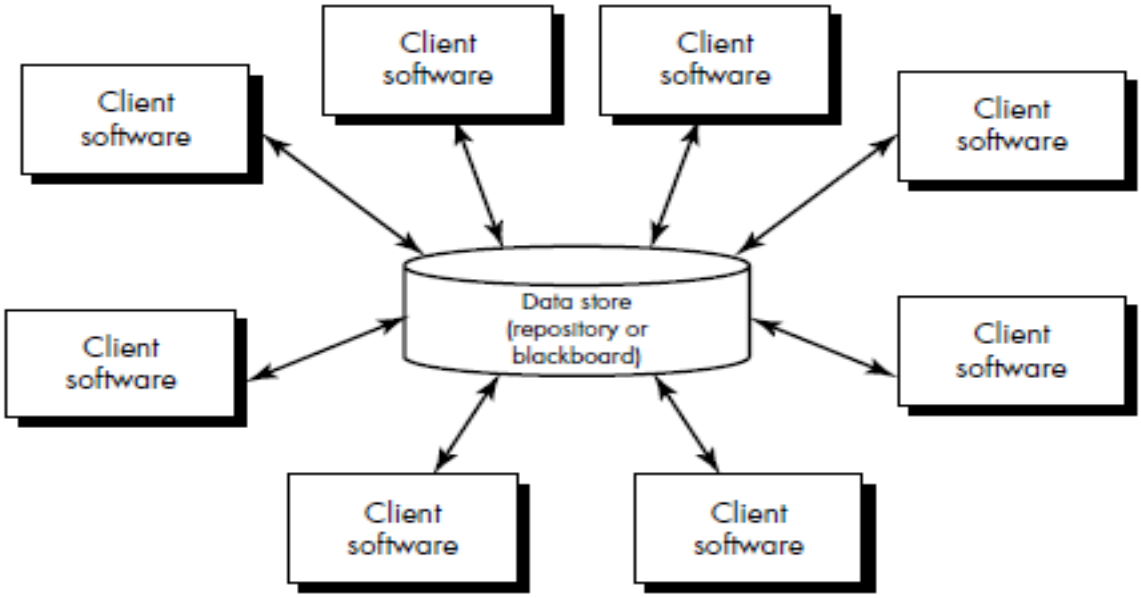


Fig.3.15. Data-centered architecture

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository.
- client software accesses the data independent of any changes to the data or the actions of other client software.
- Existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently)

Data-flow architectures

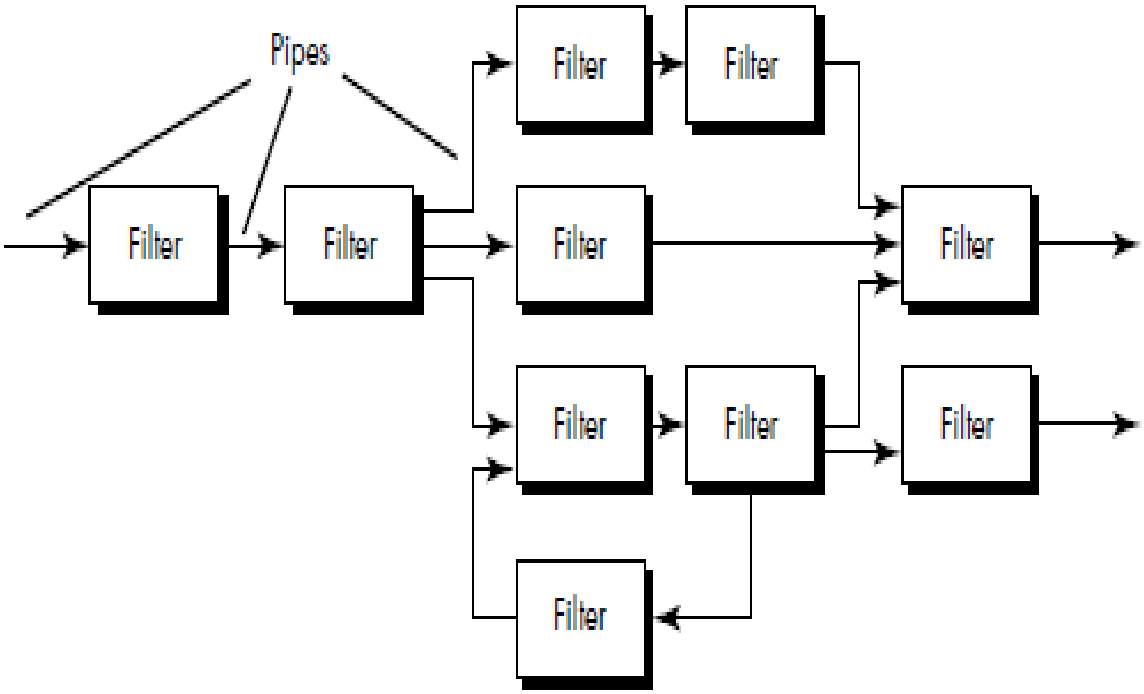


Fig.3.16. Data-flow architecture

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

Call and return architectures

- This architectural style enables a software designer to achieve a program structure that is relatively easy to modify and scale.

- **Main program/subprogram architectures**. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.
- **Remote procedure call architectures**. The components of a main program/ subprogram architecture are distributed across multiple computers on a network

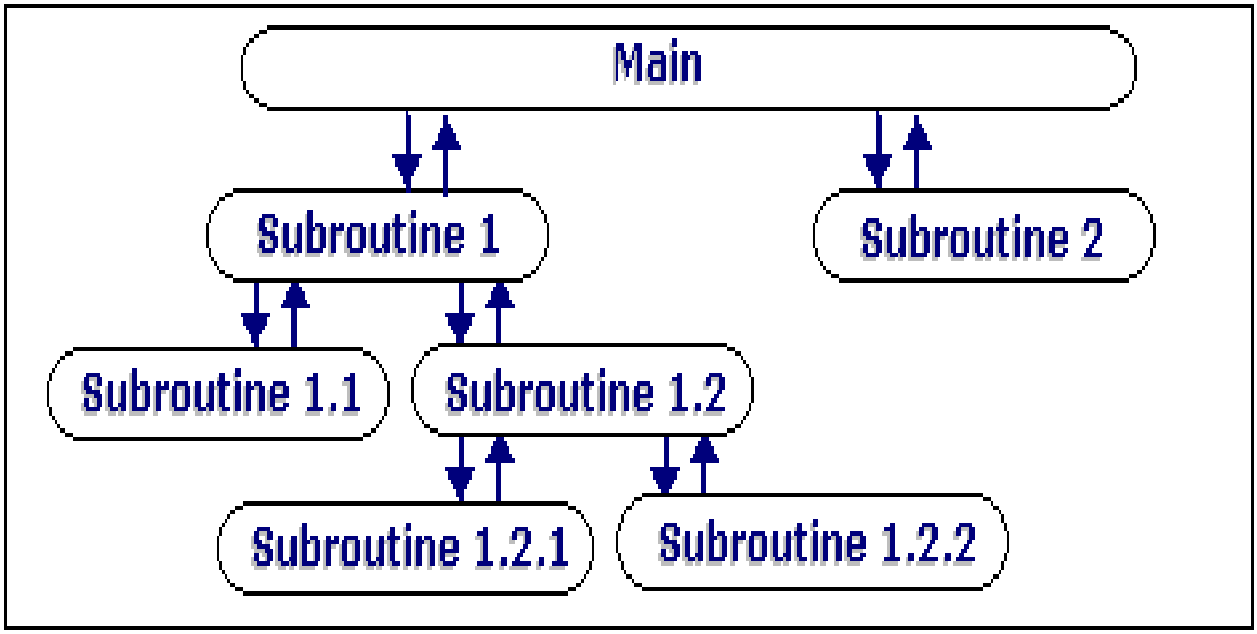


Fig.3.17. Call and return architecture

Object-oriented architectures

- The components of a system encapsulate data and the operations that must be applied to manipulate the data.
- Communication and coordination between components is accomplished via message Passing.

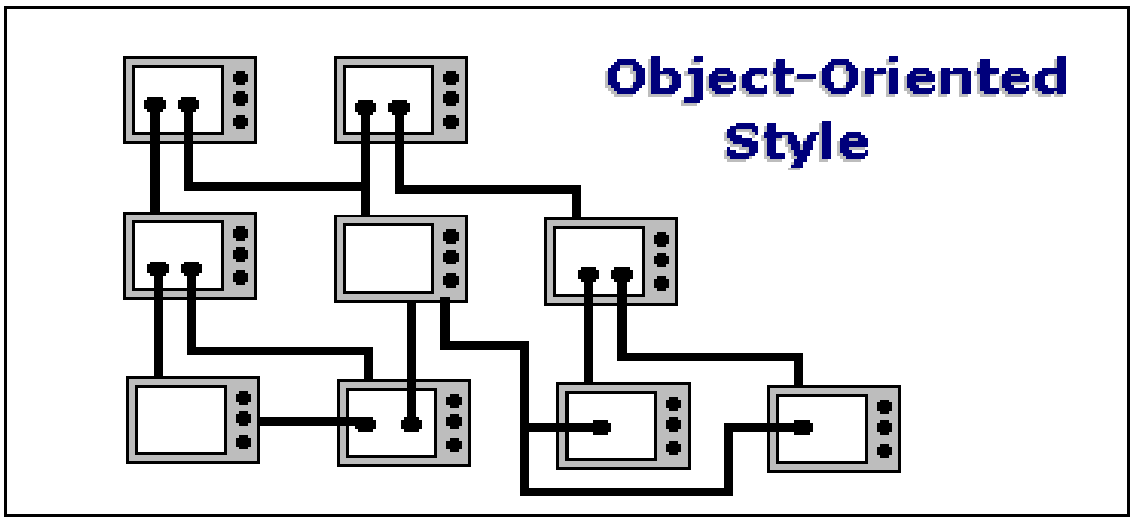


Fig.3.18. Object-oriented architecture

Layered architectures

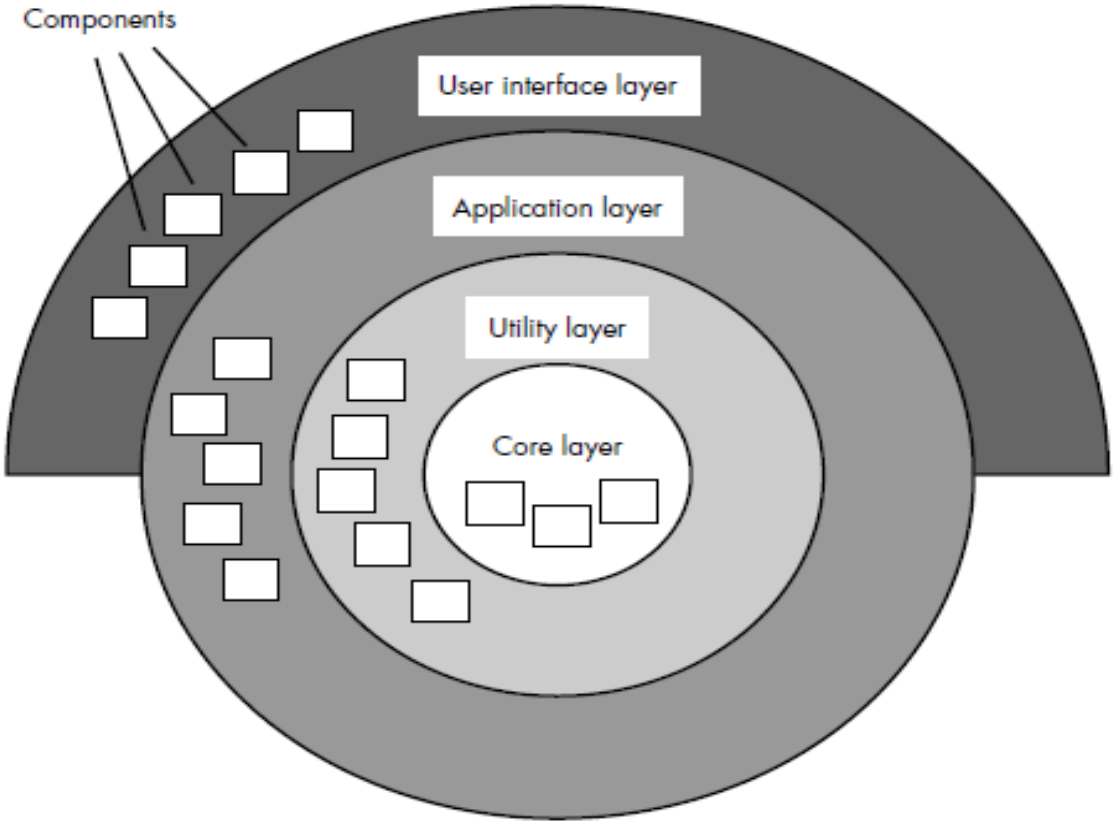


Fig.3.19. Layered architecture

- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.

DESIGN DOCUMENTATION

- First, the **overall scope of the design effort** is described. Much of the information presented here is derived from the System Specification and the analysis model (Software Requirements Specification).
- Next, the **data design** is specified. Database structure, any external file structures,internal data structures, and a cross reference that connects data objects to specific files are all defined
- The **architectural design** indicates how the program architecture has been derived from the analysis model. In addition, structure charts are used to represent the module hierarchy
- The **design of external and internal program interfaces** is represented and a **detailed design of the human/machine interface** is described. In some cases, a detailed prototype of a GUI may be represented.
- **Components**—separately addressable elements of software such as subroutines, functions, or procedures—are initially described with an English-language processing narrative.
- The **processing narrative** explains the procedural function of a component (module). Later, a procedural design tool is used to translate the narrative into a structured description.
- The Design Specification contains a **requirements cross reference**. The purpose of this cross reference is (1) to establish that all requirements are satisfied by the software design and (2) to indicate which components are critical to the implementation of specific requirements.

- The final section of the Design Specification contains **supplementary data. Algorithm descriptions, alternative procedures, tabular data**, excerpts from other documents, and other relevant information

MAPPING REQUIREMENTS INTO A SOFTWARE ARCHITECTURE

- **Transform Flow**
- Data "continuously" moves through a collection of incoming flow processes, transform center processes, and finally outgoing flow processes.
- **Incoming flow:** Information enters the system along paths that transform external data into internal data
- **Transform flow:** Internal data is processed
- **Outgoing flow:** Internal data are transformed into external data

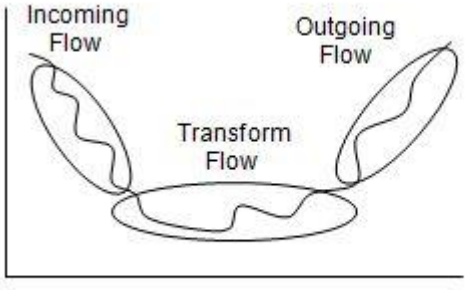


Fig.3.20. Transform Flow

Transactional Flow

Data "continuously" moves through a collection of incoming flow processes, reaches a particular transaction center process, and then follows one of anumber of actions paths. Each action path is again a collection of processes.

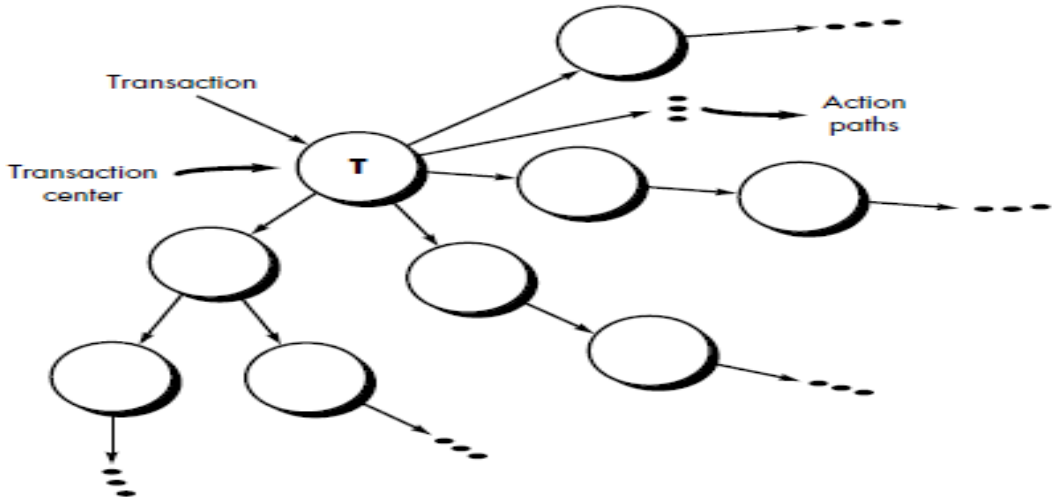


Fig.3.21. Transactional Flow

Transform Mapping

- Mapping the transform data flow diagram into software architecture design model
- Input: transform data flow diagram
- Output: software architecture

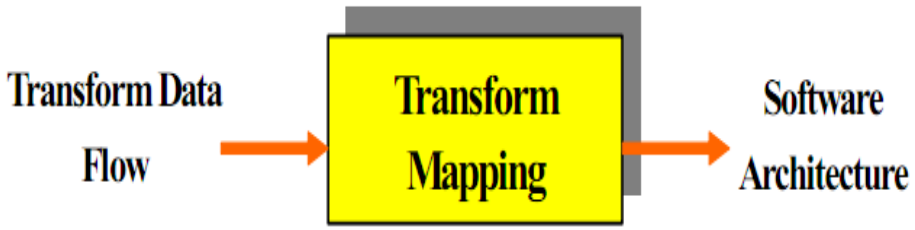


Fig.3.22. Transform Mapping

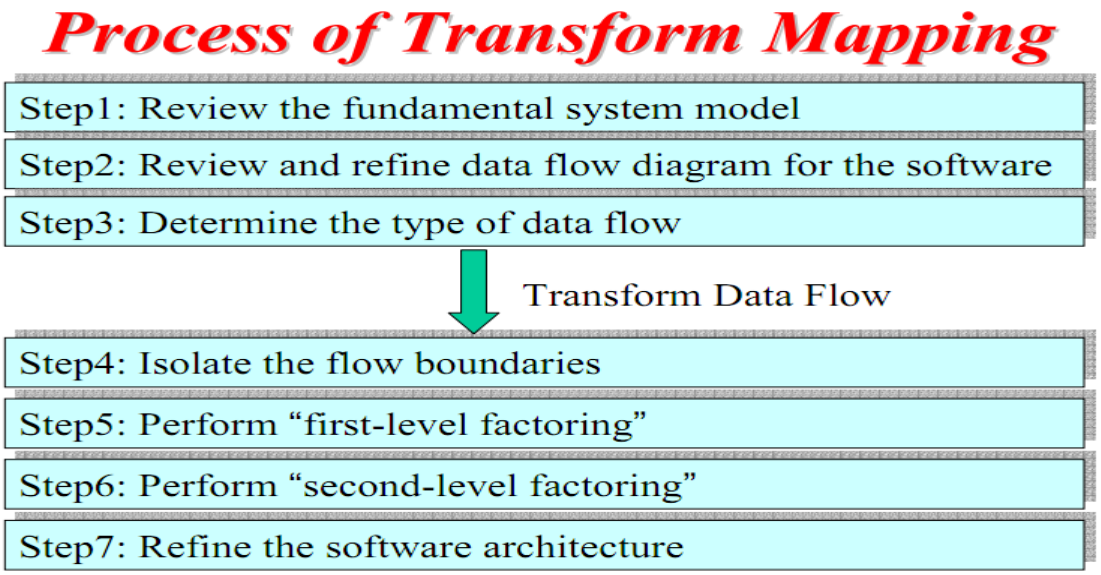


Fig.3.23. Process of Transform Mapping

Step1:Review the fundamental system model

- What is fundamental system model?

❑ Top-level or 0-level data flow diagram

- Why reviewing the fundamental system model?

❑ To evaluate the SRS in order to guarantee that the system model conforms to the real system

Level 0 DFD

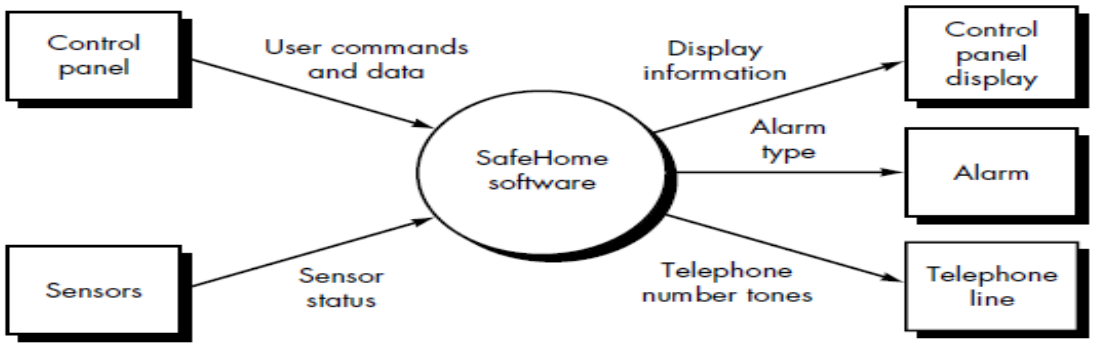


Fig.3.24. Level 0 DFD

Step2: Review and Refine Data Flow Diagram for the Software

- DFD is correct
- Produce greater detail
- Each transform in the data flow diagram exhibits relatively high cohesion that can be implemented as a component in software

Level 1 DFD

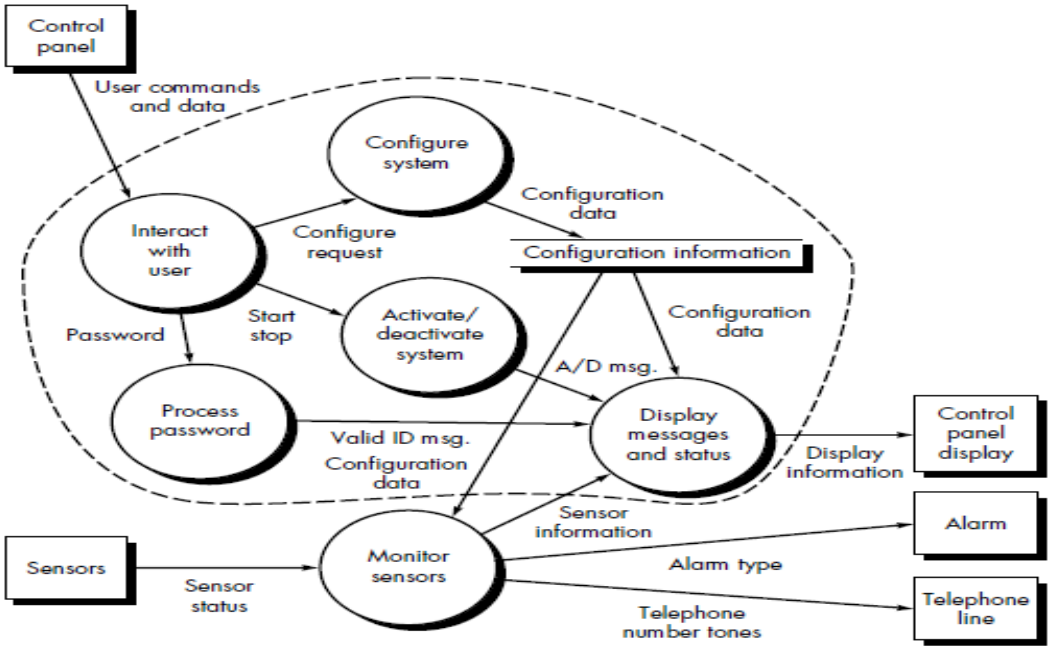


Fig.3.25. Level 1 DFD

Level 2 DFD Refine monitor sensor process

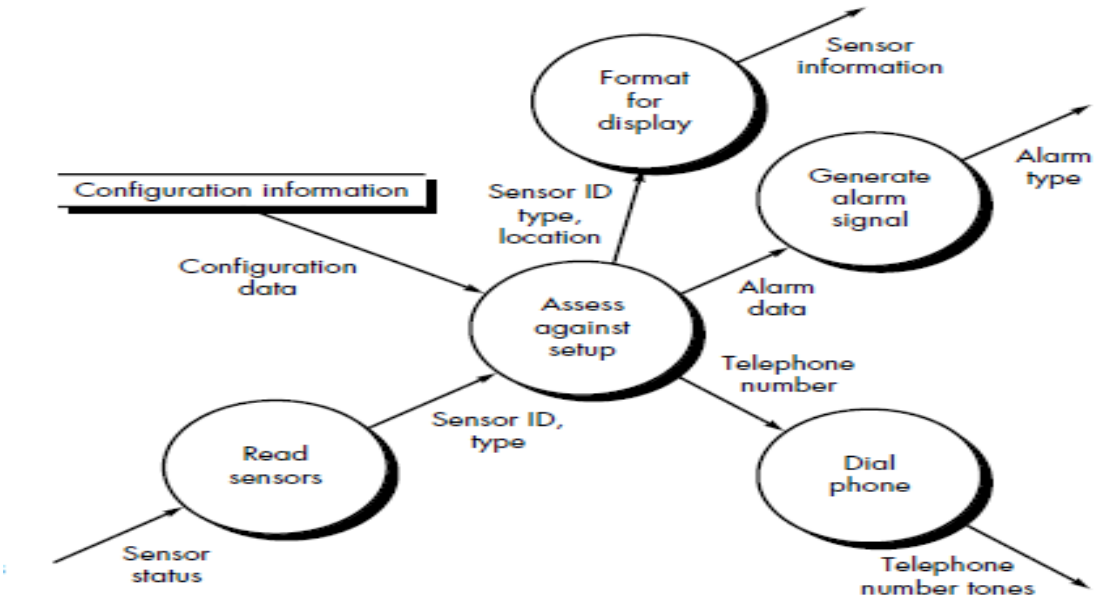


Fig.3.26. Level 2 DFD

Step 3: Determine the type of data flow

- Transform flow or transaction flow
- Different type of data flow corresponds to different mapping approach

Step 4:Isolate the flow boundaries

- Incoming flow
- Transform center
- Outgoing flow
- Different designers may select slightly different points in flow as boundary location, and therefore have different design

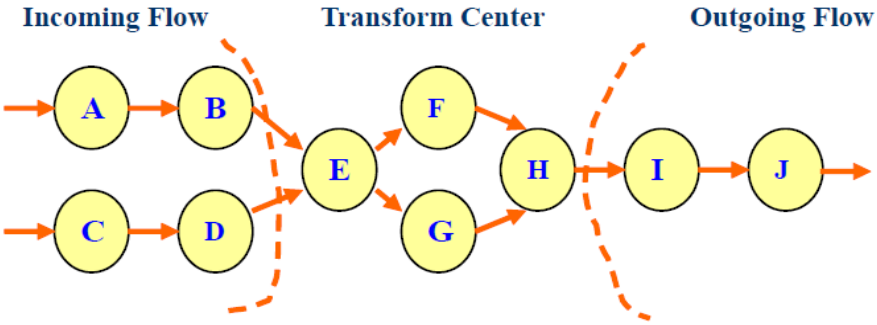


Fig.3.27. Isolate the flow boundaries

Level 3 DFD for monitor sensors with flow boundaries

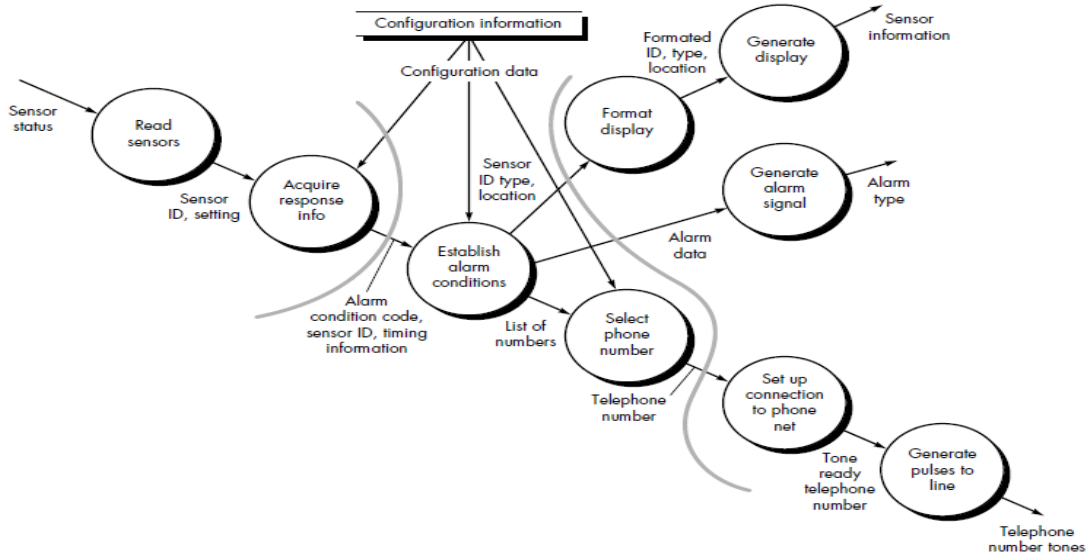


Fig.3.28. Level 3 DFD for monitor sensors with flow boundaries

Step5. Perform First-level Factoring

- **Top-level modules:** decision making
- **Middle-level modules:** some control and some work

- **Low-level modules:** perform most input, computational, and output work

- The generated software structure can be specified by hierarchy diagram or structure diagram

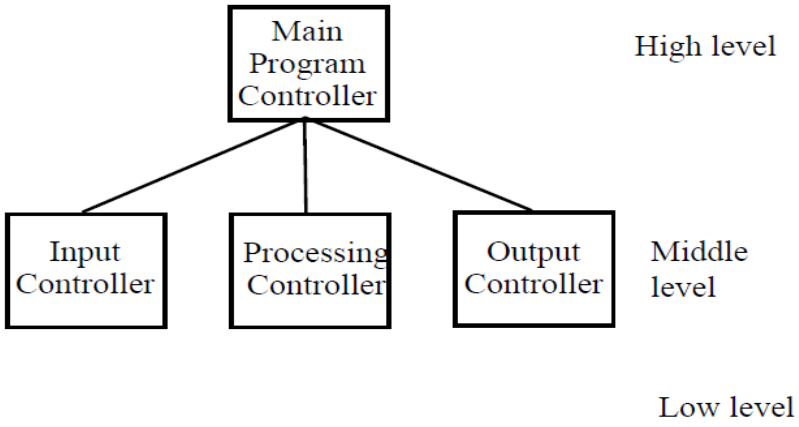


Fig.3.29. First-level Factoring

- **First-level Factoring of monitor sensor**

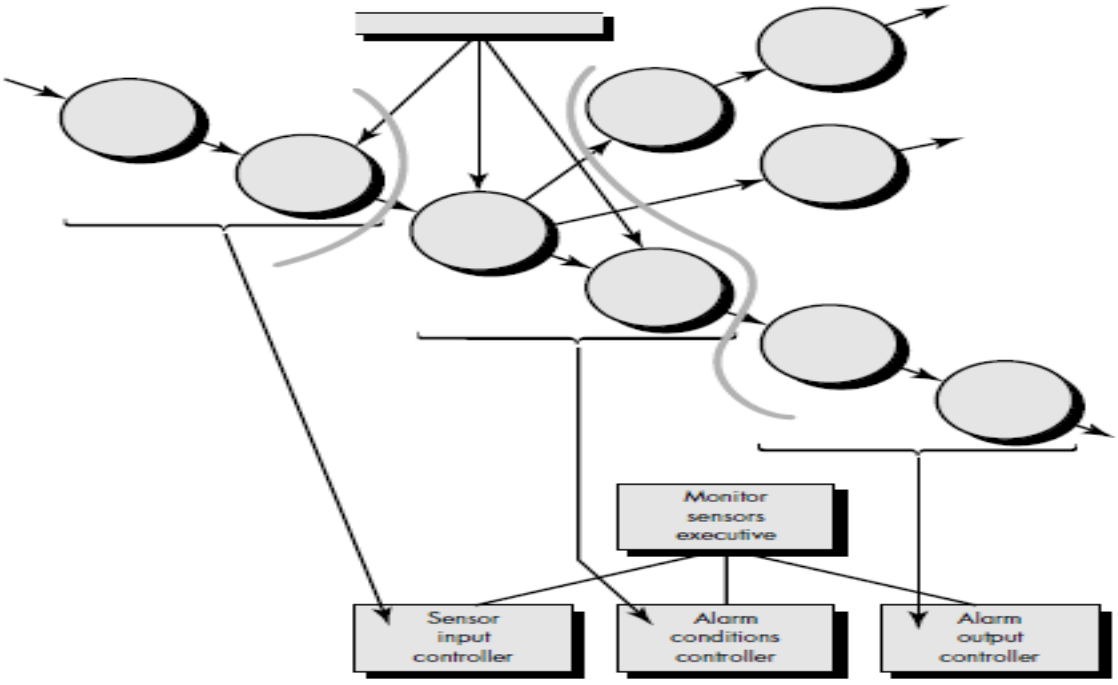


Fig.3.30. First-level Factoring of monitor sensor

Step 6. Perform second-level Factoring

- Mapping individual transforms of a DFD into aappropriate modules with the architecture
- Approach
- Beginning at the transform center boundary and moving outward along incoming and then outgoingpaths, transforms are mapped into sub-ordinate levels of the software architecture, transforms aremapped into sub-ordinate levels of the software structure
- **Second level factoring of monitor sensor**

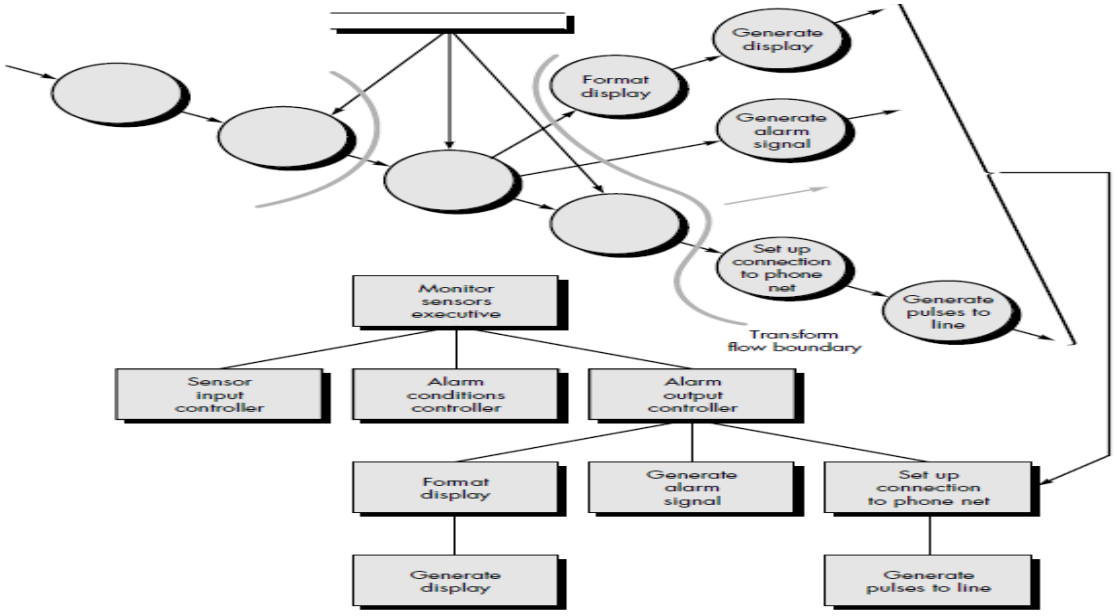


Fig.3.31. Second level factoring of monitor sensor

First iteration program structure for monitor sensor

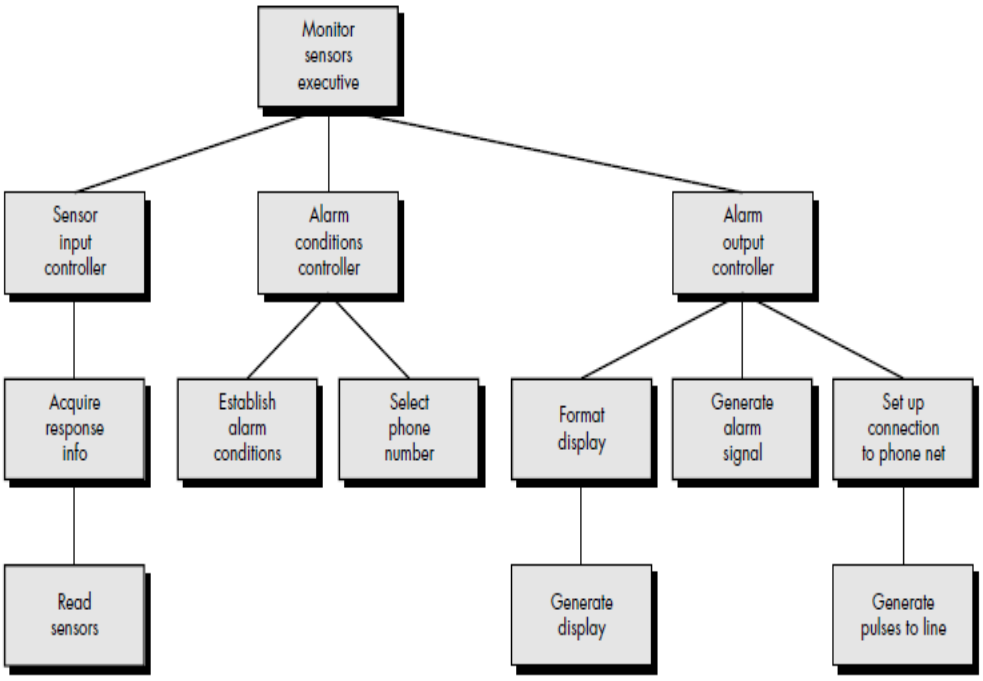


Fig.3.32. First iteration program structure for monitor sensor

Refine the software Architecture

- Applying principles of "modular"

❑ Components are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling.

❑ A good structure can be implemented without any difficulty, tested without confusion, and maintained without grief

Refined program structure

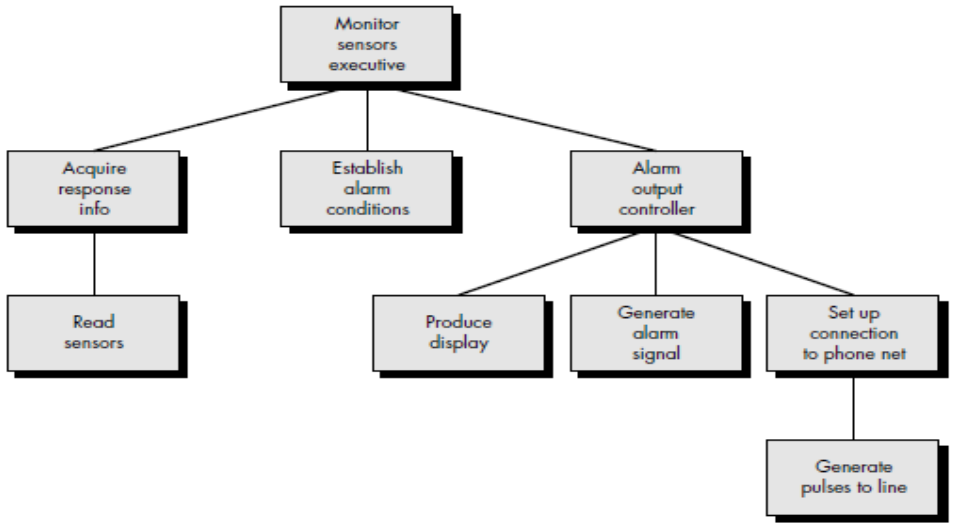


Fig.3.33. Refined program structure

Transaction Mapping

- Mapping the transaction data flow diagram into software architecture design model

❑ Input: transaction data flow diagram

❑ Output: software architecture

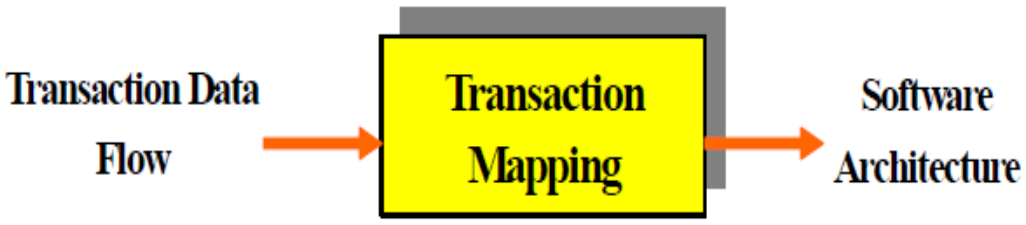


Fig.3.34. Transaction Mapping

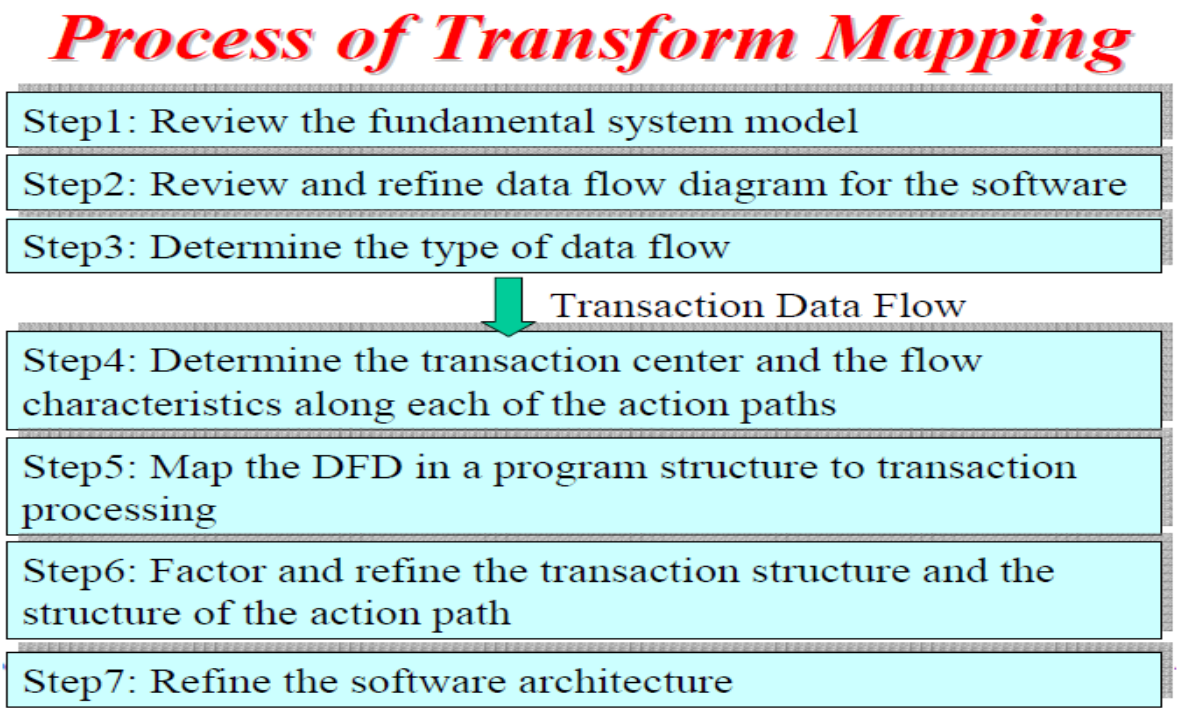


Fig.3.35. Process of Transaction Mapping

Level 1 DFD

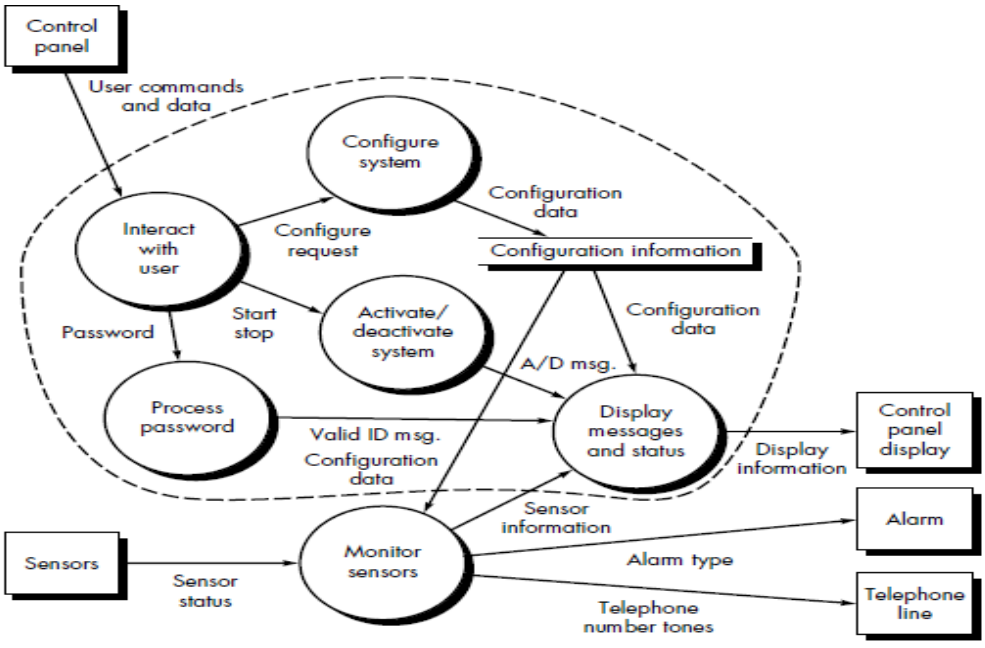


Fig.3.36. Level 1 DFD

Level 2 DFD for user interaction sub system

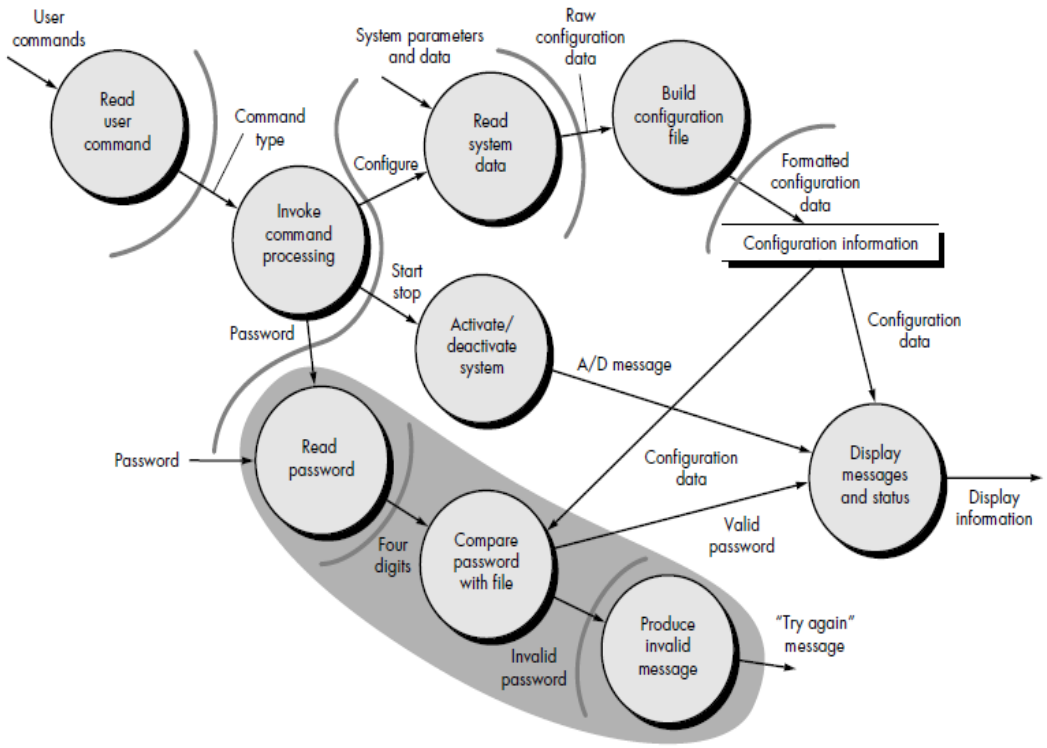


Fig.3.37. Level 2 DFD for user interaction sub system

- **Step 3. Determine whether the DFD has transform or transaction flow characteristics.**
- Steps 1, 2, and 3 are identical to corresponding steps in transform mapping.
- The DFD shown in Figure has a classic transaction flow characteristic. However, flow along two of the action paths emanating from the invoke command processing bubble appears to have transform flow characteristics.
- Flow boundaries must be established for both flow types.

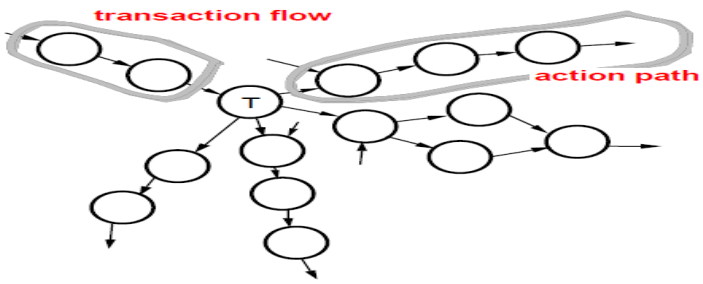


Fig.3.38. Flow boundaries

Step 4:Identify the transaction centre and action path

- Three parts of transaction DFD

❑ Input flow

❑ Transaction center

❑ Action path

- Evaluate the flow characteristics of each action path

❑ Transaction flow or transform flow

- Each action path must be evaluated for its individual flow characteristic. For example, the "password" path has transform characteristics. Incoming, transform, and outgoing flow are indicated with boundaries

Step 5: Map DFD in program structure

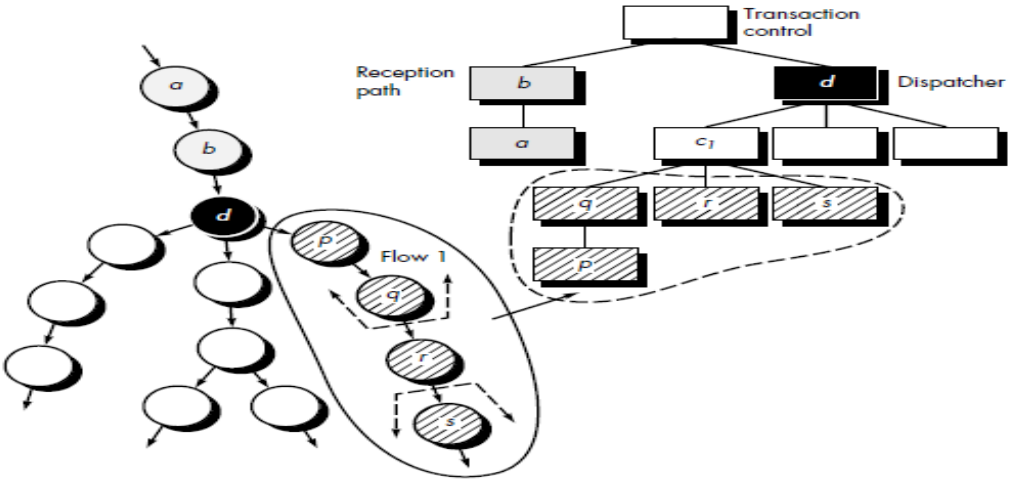


Fig.3.39. Map DFD in program structure

- Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch
- The structure of the incoming branch is developed in much the same way as transform mapping. Starting at the transaction center, bubbles along the incoming path are mapped into modules.
- The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules.
- Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics.

First level factoring for user interaction sub system

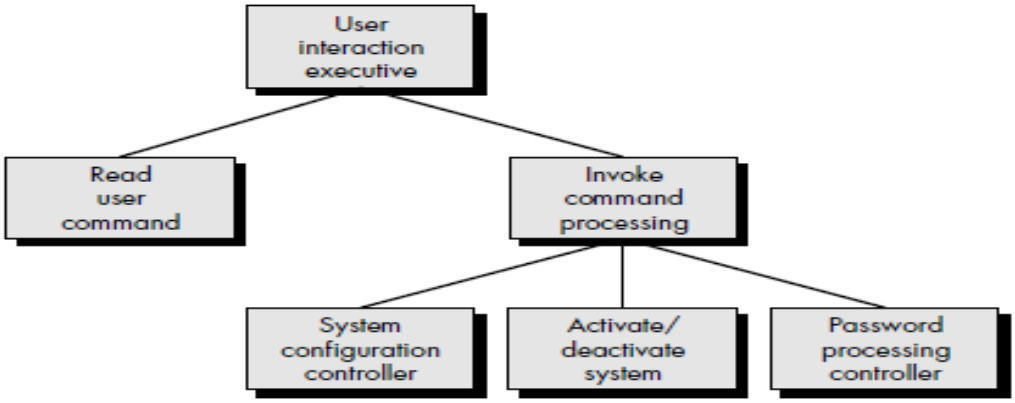


Fig.3.40. First level factoring for user interaction sub system

Step 6:Factor and Refine the Transaction Structure and Each Action Path

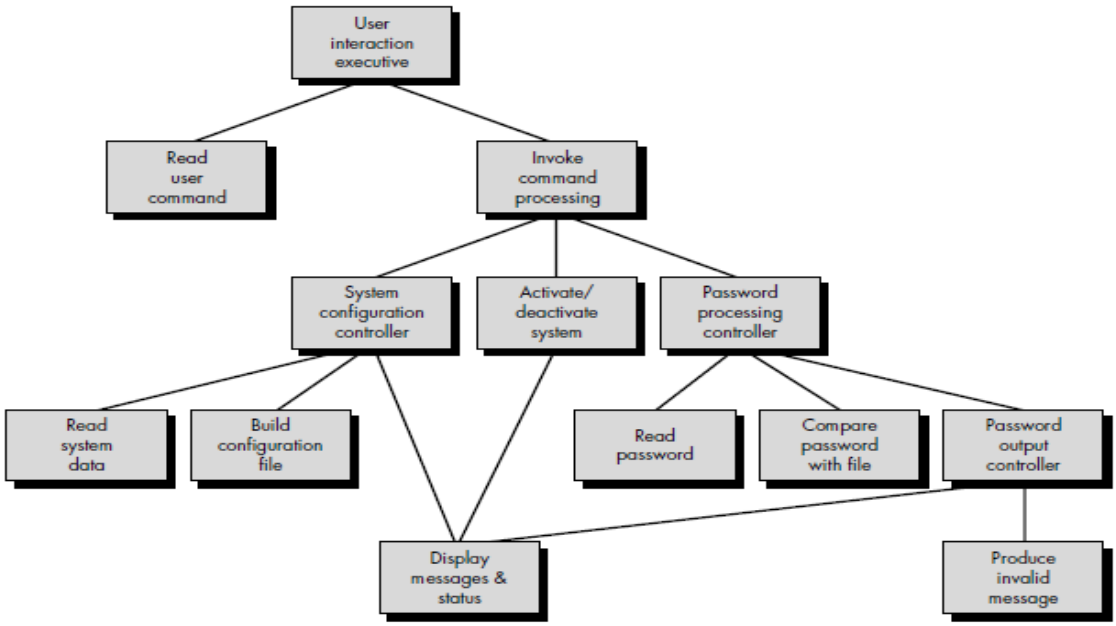


Fig.3.41. Factor and Refine the Transaction Structure and Each Action Path

- **Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.**

USER INTERFACE DESIGN

User Interface Design Models

- **User model**—a profile of all end users of the system
- **Design model**—a design realization of the user model
- **Mental model** (system perception)—the user's mental image of what the interface is
- **Implementation model**—the interface "look and feel" coupled with supporting information that describe interface syntax and semantics

User Interface Design Process

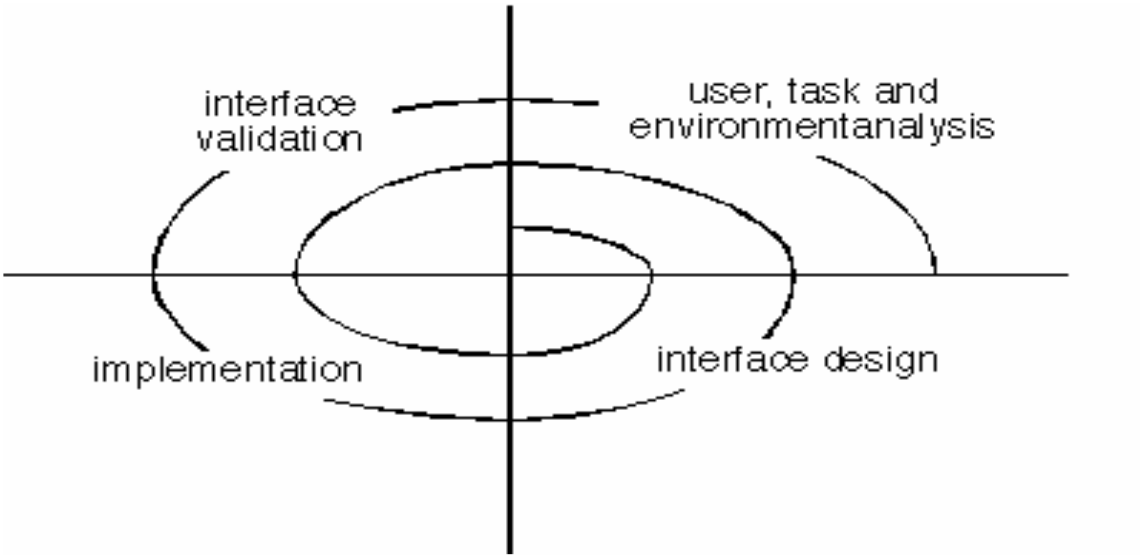


Fig.3.42. User Interface Design Process

Interface Analysis

- Interface analysis means understanding
- (1) the people (end-users) who will interact with the system through the interface;
- (2) the tasks that end-users must perform to do their work,
- (3) the content that is presented as part of the interface
- (4) the environment in which these tasks will be conducted**.**

User Analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?

- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology the sits behind the interface

Task Analysis and Modeling

- Answers the following questions…

➢ What work will the user perform in specific circumstances?

➢ What tasks and subtasks will be performed as the user does the work?

➢ What specific problem domain objects will the user manipulate as work is performed?

➢ What is the sequence of work tasks—the workflow?

➢ What is the hierarchy of tasks?

➢ Use-cases define basic interaction

➢ Task elaboration refines interactive tasks

➢ Object elaboration identifies interface objects (classes)

➢ Workflow analysis defines how a work process is completed

➢ when several people (and roles) are involved

INTERFACE DESIGN ACTIVITIES

- The first interface design steps can be accomplished using the following approach:

- ❑ Establish the goals and intentions for each task.
- ❑ Map each goal and intention to a sequence of specific actions.
- ❑ Specify the action sequence of tasks and subtasks, also called a user scenario, as it will be executed at the interface level.
- ❑ Indicate the state of the system; that is, what does the interface look like at the time that a user scenario is performed?
- ❑ Define control mechanisms; that is, the objects and actions available to the user to alter the system state.
- ❑ Show how control mechanisms affect the state of the system.
- ❑ Indicate how the user interprets the state of the system from information provided through the interface.

IMPLEMENTATION TOOLS

- Called user- interface toolkits or user-interface development systems (UIDS), these tools provide components or objects that facilitate creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment.
- A UIDS provides built-in mechanisms for

- ❑ managing input devices (such as a mouse or keyboard)
- ❑ validating user input
- ❑ handling errors and displaying error messages
- ❑ providing feedback (e.g., automatic input echo)
- ❑ providing help and prompts
- ❑ handling windows and fields, scrolling within windows
- ❑ establishing connections between application software and the interface
- ❑ insulating the application from interface management functions
- ❑ allowing the user to customize the interface

DESIGN EVALUATION

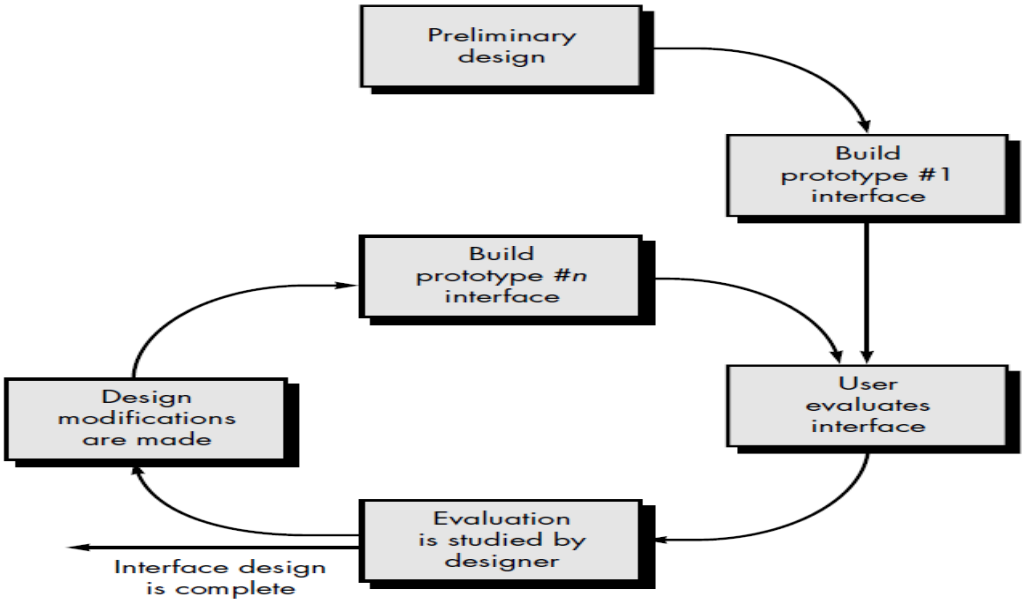


Fig.3.43. Design Evaluation

Jackson System Development (JSD) is a method of system development that covers the software life cycle either directly or by providing a framework into which more specialized techniques can fit. JSD can start from the stage in a project when there is only a general statement of requirements.

However many projects that have used JSD actually started slightly later in the life cycle, doing the first steps largely from existing documents rather than directly with the users.

Phases of JDS:

JSD has 3 phases:

Modelling Phase:

In the modelling phase of JSD the designer creates a collection of entity structure diagrams and identifies the entities in the system, the actions they perform, the attributes of the actions and time ordering of the actions in the life of the entities.

Specification Phase:

This phase focuses on actually what is to be done? Previous phase provides the basic for this phase. An sufficient model of a time-ordered world must itself be time-ordered. Major goal is to map progress in the real world on progress in the system that models it.

Implementation Phase:

In the implementation phase JSD determines how to obtain the required functionality. Implementation way of the system is based on transformation of specification into efficient set of processes. The processes involved in it should be designed in such a manner that it would be possible to run them on available software and hardware.

JSD Steps:

Initially there were six steps when it was originally presented by Jackson, they were as below:

- Entity/action step
- Initial model step
- Interactive function step
- Information function step
- System timing step
- System implementation step

Later some steps were combined to create method with only three steps:

- Modelling Step
- Network Step
- Implementation Step

Merits of JSD:

- It is designed to solve real time problem.
- JSD modelling focuses on time.

- ❖ It considers simultaneous processing and timing.
- ❖ It is a better approach for micro code application.

Demerits of JSD:

- ❖ It is a poor methodology for high level analysis and data base design.
- ❖ JSD is a complex methodology due to pseudo code representation.
- ❖ It is less graphically oriented as compared to SA/SD or OMT.
- ❖ It is a bit complex and difficult to understand.

Design for reuse

- Design reuse is the process of building new software applications and tools by reusing previously developed designs. New features and functionalities may be added by incorporating minor changes.
- Design reuse involves the use of designed modules, such as logic and data, to build a new and improved product.
- The reusable components, including code segments, structures, plans and reports, minimize implementation time and are less expensive. This avoids reinventing existing software by using techniques already developed and to create and test the software.
- Design reuse involves many activities utilizing existing technologies to cater to new design needs.
- The ultimate goal of design reuse is to help the developers create better products maximizing it's value with minimal resources, cost and effort.
- Today, it is almost impossible to develop an entire product from scratch. Reuse of design becomes necessary to maintain continuity and connectivity.

- In the software field, the reuse of the modules and data helps save implementation time and increases the possibility of eliminating errors due to prior testing and use.
- Design reuse requires that a set of designed products already exist and the design information pertaining to the product is accessible.
- Large software companies usually have a range of designed products. Hence the reuse of design facilitates making new and better software products.
- Many software companies have incorporated design reuse and have seen considerable success.
- The effectiveness of design reuse is measured in terms of production, time, cost and quality of the product. These key factors determine whether a company has been successful in making design reuse a solution to its new software needs and demands.
- With proper use of existing technology and resources, a company can benefit in terms of cost, time, performance and product quality.
- A proper process requires an intensive design reuse process model. There are two interrelated process methodologies involved in the systematic design reuse process model.

The data reuse process is as follows:

1. Gathering Information: This involves the collection of information, processing and modeling to fetch related data.
2. Information Reuse: This involves the effective use of data.

The design reuse process has four major issues:

1. Retrieve
2. Reuse
3. Repair
4. Recover

These are generally referred to as the four Rs. In spite of these challenges, companies have used the design reuse concept as a successfully implemented concept in the software field at different levels, ranging from low level code reuse to high level project reuse.

Software Configuration Management

• Why Software Configuration Management ?

• The problem:

– Multiple people have to work on software that is changing

– More than one version of the software has to be supported:

• Released systems

• Custom configured systems (different functionality)

• System(s) under development

– Software must run on different machines and operating systems

Need for coordination

• Software Configuration Management

– manages evolving software systems

– controls the costs involved in making changes to a system

• **Definition:**

– A set of management disciplines within the software engineering process to develop a baseline.

- **Description:**

– Software Configuration Management encompasses the disciplines and techniques of initiating, evaluating and controlling change to software products during and after the software engineering process.

SCM Activities

• Configuration item identification

– modeling of the system as a set of evolving components

• Promotion management

– is the creation of versions for other developers

• Release management

– is the creation of versions for the clients and users

• Branch management

– is the management of concurrent development

• Variant management

– is the management of versions intended to coexist

• Change management

– is the handling, approval and tracking of change requests

SCM Roles

• **Configuration Manager**

– Responsible for identifying configuration items. The configuration manager can also be responsible for defining the procedures for creating promotions and releases

• **Change control board member**

– Responsible for approving or rejecting change requests

• **Developer**

– Creates promotions triggered by change requests or the normal activities of development. The developer checks in changes and resolves conflicts

Auditor

– Responsible for the selection and evaluation of promotions for release and for ensuring the consistency and completeness of this release

Terminology and Methodology

– Configuration Items

– Baselines

– SCM Directories

– Versions, Revisions and Releases

Configuration Item

"An aggregation of hardware, software, or both, that is designated for configuration management and treated as a single entity in the configuration management process."

▪ Software configuration items are not only program code segments but all type of documents according to development, e.g

▪ all type of code files

▪ drivers for tests

▪ analysis or design documents

▪ user or developer manuals

▪ system configurations (e.g. version of compiler used)

▪ In some systems, not only software but also hardware configuration items (CPUs, bus speed frequencies) exist!

• Large projects typically produce thousands of entities (files, documents, ...) which must be uniquely identified.

But not every entity needs to be configured all the time. Issues:

– What: Selection of CIs (What should be managed?)

– When: When do you start to place an entity under configuration control?

– Starting too early introduces too much bureaucracy

▪ Starting too late introduces chaos

▪ Some of these entities must be maintained for the lifetime of the software. This includes

also the phase, when the software is no longer developed but still in use; perhaps by industrial customers who are expecting proper support for lots of years.

▪ An entity naming scheme should be definedso that related documents have related names.

▪ Selecting the right configuration items is a skill that takes practice

▪ Very similar to object modeling

▪ Use techniques similar to object modeling for finding Cis

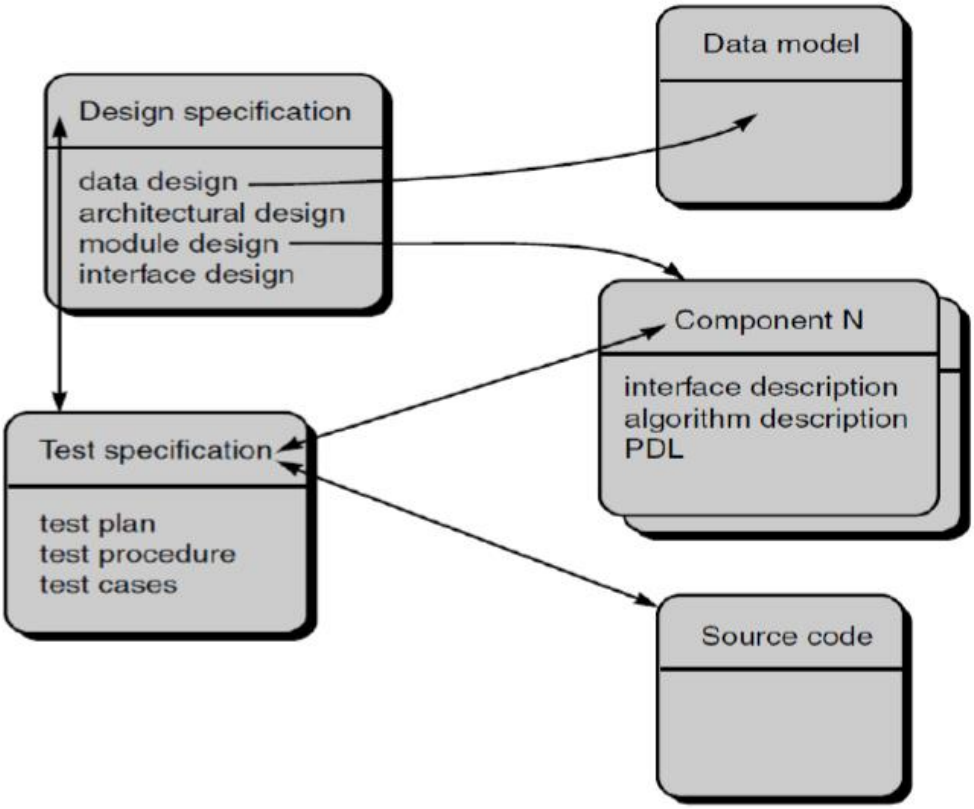


Fig.3.44. Configuration Objects

Baseline

A specification or product that has been formally reviewed and agreed to by responsible management, that thereafter serves as the basis for further development, and can be changed only through formal change control procedures."

Examples:

Baseline A: The API of a program is completely defined; the bodies of the methods are empty.

Baseline B: All data access methods are implemented and tested; programming of the GUI can start.

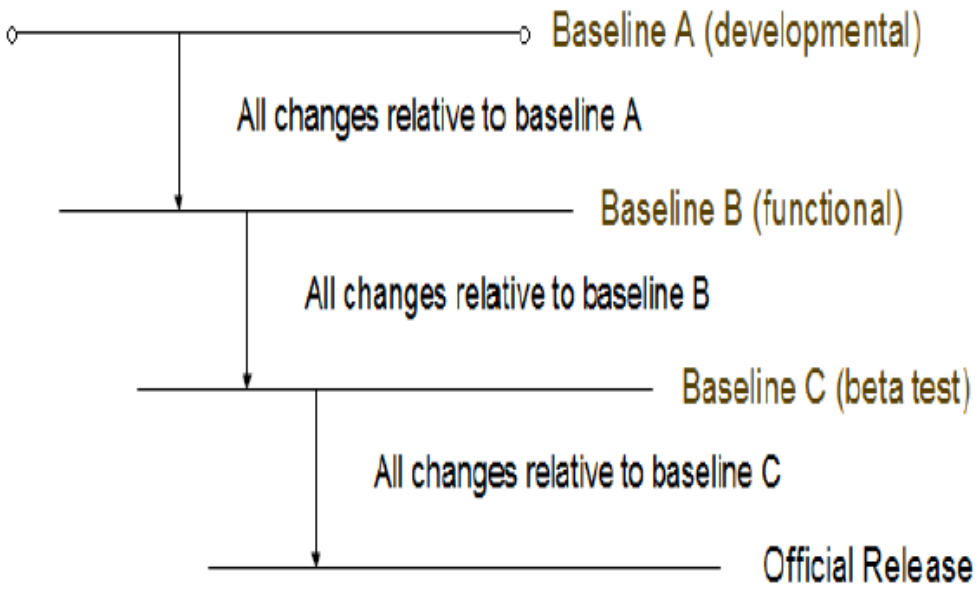
Baseline C: GUI is implemented, test-phase can start.

• As systems are developed, a series of baselines is developed, usually after a review (analysis review, design review, code review, system testing, client acceptance, ...)

– Developmental baseline (RAD, SDD, Integration Test, ...)

• Goal: Coordinate engineering activities.

Many naming scheme for baselines exist (1.0, 6.01a, ...)



SCM Directories

• Programmer's Directory (IEEE: Dynamic Library)

– Library for holding newly created or modified software entities. The programmer's workspace is controlled by the programmer only.

• Master Directory (IEEE: Controlled Library)

– Manages the current baseline(s) and for controlling changes made to them. Entry is controlled, usually after verification. Changes must be authorized.

• Software Repository (IEEE: Static Library)

– Archive for the various baselines released for general use. Copies of these baselines may be made available to requesting organizations.

Change management

• Change management is the handling of change requests

– A change request leads to the creation of a new release

• General change process

– The change is requested (this can be done by anyone including users and developers)

– The change request is assessed against project goals

– Following the assessment, the change is accepted or rejected

– If it is accepted, the change is assigned to a developer and implemented

The implemented change is audited.

• The complexity of the change management process varies with the project. Small projects can perform change requests informally and fast while complex projects require detailed change request forms and the official approval by one more managers.

Controlling Changes

• Two types of controlling change:

– Promotion: The internal development state of a software is changed.

– **Release:** A set of promotions is distributed outside the development organization.

• Approaches for controlling change to libraries (Change Policy)

– Informal (good for research type environments)

– Formal approach (good for externally developed CIs and for releases)

Change Policies

• Whenever a promotion or a release is performed, one or more policies apply. The purpose of change policies is to guarantee that each version, revision or release conforms to commonly accepted criteria.

• Examples for change policies:

"No developer is allowed to promote source code which cannot be compiled without errors and warnings."

"No baseline can be released without having been beta-tested by at least 500 external persons."

Version: Version vs. Revision vs. Release

• **Version: Version vs. Revision vs. Release**

– An initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item. Different versions have different functionality.

• **Revision:**

– Change to a version that corrects only errors in the design/code, but does not affect the documented functionality.

• **Release:**

– The formal distribution of an approved version.

SCM planning

• Software configuration management planning starts during the early phases of a project.

• The outcome of the SCM planning phase is the Software Configuration Management Plan (SCMP) which might be extended or revised during the rest of the project.

• The SCMP can either follow a public standard like the IEEE 828, or an internal (e.g. company specific) standard.

The Software Configuration Management Plan

• Defines the types of documents to be managed and a document naming scheme.

• Defines who takes responsibility for the CM procedures and creation of baselines.

• Defines policies for change control and version management.

• Describes the tools which should be used to assist the CM process and any limitations on their use.

• Defines the configuration management database used to record configuration information.

Outline of a Software Configuration Management Plan

1. Introduction

Describes purpose, scope of application, key terms and references

2. Management (WHO?)

Identifies the responsibilities and authorities for accomplishing the planned configuration management activities

3. Activities (WHAT?)

Identifies the activities to be performed in applying to the project.

4. Schedule (WHEN?)

Establishes the sequence and coordination of the SCM activities with project mile stones.

5. Resources (HOW?)

Identifies tools and techniques required for the implementation of the SCMP

6. Maintenance

Identifies activities and responsibilities on how the SCMP will be kept current during the life-cycle of the project.

Tools for Software Configuration Management

Software configuration management is normally supported by tools with different functionality.

• Examples:

– RCS

• very old but still in use; only version control system

– CVS

• based on RCS, allows concurrent working without locking

– Perforce

• Repository server; keeps track of developer's activities

– ClearCase

• Multiple servers, process modeling, policy check mechanisms

Programming standards

- Coding standards are guidelines for code style and documentation.
- They may be formal (IEEE) standards, or company specific standards.
- The aim is that everyone in the organization will be able to read and work on the code.
- Coding standards cover a wide variety of areas:
  - Program design
  - Naming conventions
  - Formatting conventions
  - Documentation
  - Use (or not) of language specific features

- Why bother with a coding standard?
  - Consistency between developers
  - Ease of maintenance and development
  - Readability, usability
- Example should make this obvious!
- No standard is perfect for every application.
  - If you deviate from the standard for any reason, document it!

Coding style

- There are several examples of coding styles. Often they differ from company to company.
- They typically have the following in common:
  - Names
    - Use full English descriptors
    - Use mixed case to make names readable
    - Use abbreviations sparingly and consistently
    - Avoid long names
    - Avoid leading/trailing underscores
  - Documentation
    - Document the purpose of every variable
    - Document why something is done, not just what
  - Member function documentation
    - What & why member function does what it does
    - Parameters/return value
    - How function modifies object
    - Preconditions/postconditions

        - Concurrency issues
        - Restrictions
    - Document why the code does things as well as what it does.

Rules

- Coding standards need not be onerous – find out what works for your organization/team and stick to it.
- Standardize early – the cost of retrofitting a standard is prohibitive.
- Encourage a culture where a standard is followed.
- The more commonly accepted the standard is, the easier it is for the team members to communicate.
- Invent standards where necessary, but do not waste time creating a standard you won't ever use again!
- All languages have recommended coding standards available. It is worthwhile finding out about these industry standards.
- Push for organizational standards wherever possible.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-IV- SOFTWARE ENGINEERING– SBS1204

SBS1204 - SOFTWARE ENGINEERING
UNIT 4

Levels - Test activities - Types of s/w test - Black box testing - Testing boundary condition - Structural testing- Test coverage criteria based on data flow mechanisms - Regression testing - Testing in the large- S/W testing strategies - Strategic approach and issues - Unit testing - Integration testing - Validation testing - System testing and debugging. Case studies – Writing black box and white box testing

Software Testing Fundamentals

- Software Testing is the process of executing a program or system with the intent of finding errors
- Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results
- Testing software is operating the software under controlled conditions, to (1) verify that it behaves -as specified‖; (2) to detect errors, and (3) to validate that what has been specified is what the user actually wanted.

Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error

Testing Principles

- All tests should be traceable to customer requirements.
- Tests should be planned long before testing begins.
- The Pareto principle applies to software testing
- Testing should begin -in the small‖ and progress toward testing -in the large.
- Exhaustive testing is not possible
- To be most effective, testing should be conducted by an independent third party.

Testability

- Software testability is simply how easily [a computer program] can be tested
- The checklist that follows provides a set of characteristics that lead to testable software.
- **Operability.** "The better it works, the more efficiently it can be tested.”
- **Observability.** "What you see is what you test."
- **Controllability.** "The better we can control the software, the more the testing can be automated and optimized."

- **Decomposability.** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."
- **Simplicity.** "The less there is to test, the more quickly we can test it."
- **Stability.** "The fewer the changes, the fewer the disruptions to testing."
- **Understandability.** "The more information we have, the smarter we will test."

White-Box Testing

- White-box testing, sometimes called glass-box testing, is a test case design method that uses the control structure of the procedural design to derive test cases.
- Using white-box testing methods, the software engineer can derive test cases that
  - guarantee that all independent paths within a module have been exercised at least once,
  - exercise all logical decisions on their true and false sides,
  - execute all loops at their boundaries and within their operational bounds, and
  - exercise internal data structures to ensure their validity

Basis Path Testing

- The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

Flow Graph Notation

- The flow graph depicts logical control flow ,Each structured construct has a corresponding flow graph symbol.

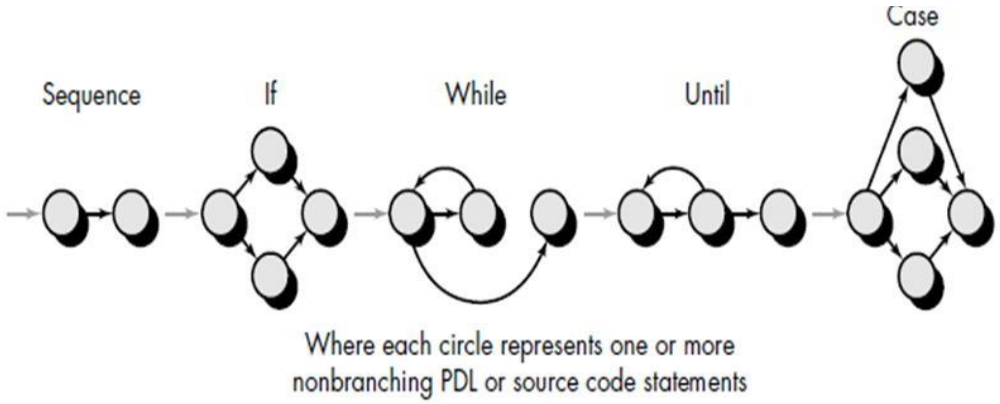


Fig.4.1. Flow Graph Notation

Cyclomatic Complexity

- Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program
- When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program

Flow chart

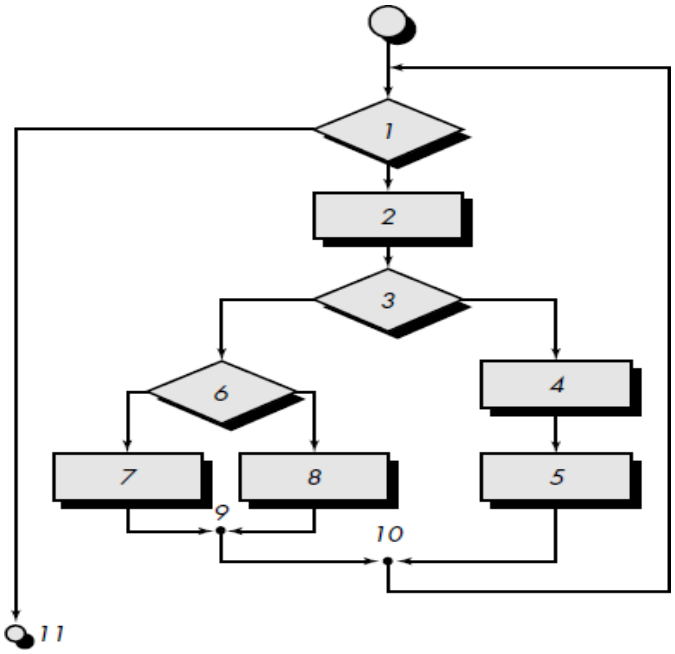


Fig.4.2. Flow chart

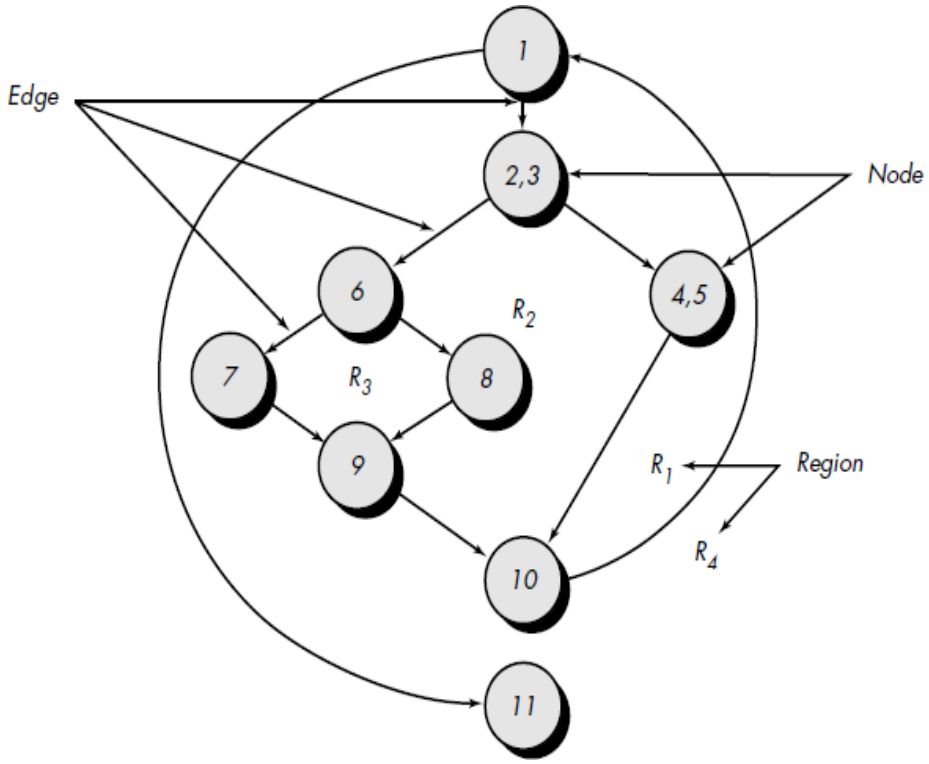


Fig.4.3. Flow Graph

Paths

- path 1: 1-11
- path 2: 1-2-3-4-5-10-1-11
- path 3: 1-2-3-6-8-9-10-1-11
- path 4: 1-2-3-6-7-9-10-1-11

Cyclomatic Complexity

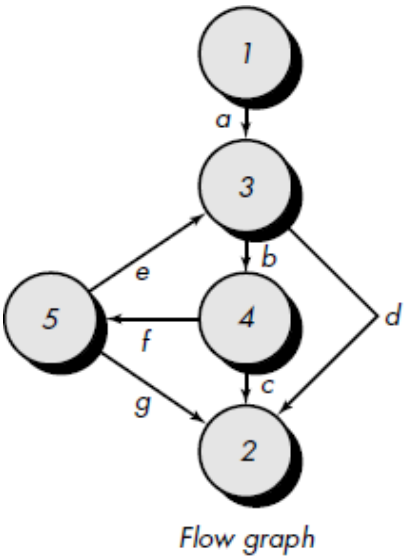
- The number of regions of the flow graph correspond to the cyclomatic complexity.
  - **The flow graph has four regions.**
- Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E - N + 2 where E is the number of flow graph edges, N is the number of flow graph nodes.
  - **V(G) = 11 edges - 9 nodes + 2 = 4.**
- Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P + 1 where P is the number of predicate nodes contained in the flow graph G.
  - **V(G) = 3 predicate nodes + 1 = 4**

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.

- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- A simple example of a flow graph and its corresponding graph matrix

Graph matrix



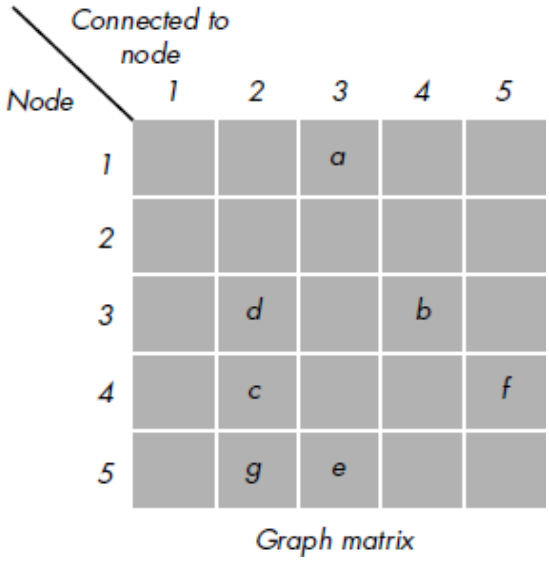


Fig.4.4. Flow Graph **Fig.4.5.Graph Matrix**

Connection matrix

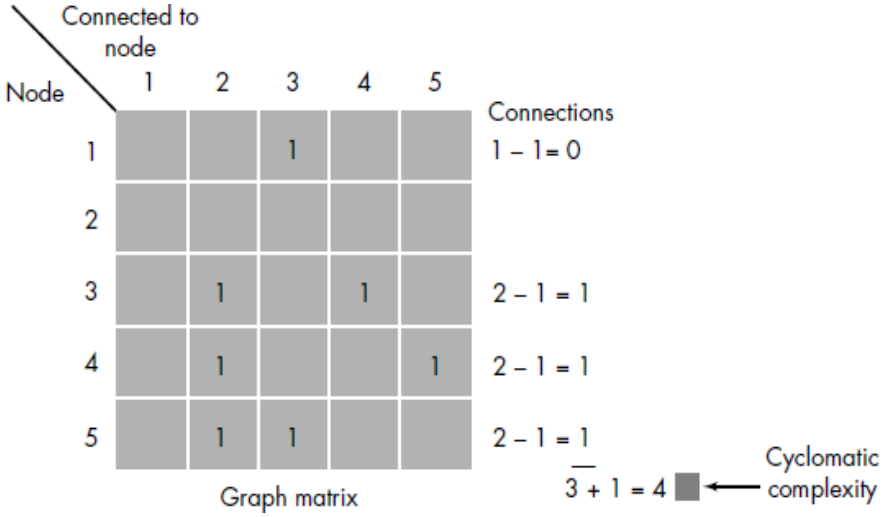


Fig.4.6 Connection matrix

- Each node on the flow graph is identified by numbers, while each edge is identified by letters.

- A letter entry is made in the matrix to correspond to a connection between two nodes. node 3 is connected to node 4 by edge b
- The graph matrix is nothing more than a tabular representation of a flow graph. by adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight provides additional information about control flow. form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).
- Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity).Represented in this form, the graph matrix is called a connection matrix.
- Each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity

Control Structure Testing

Condition Testing

- Condition testing is a test case design method that exercises the logical conditions contained in a program module
- A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form
- E1 <relational-operator> E2
- where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥.
- A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses

Types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error

Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program

- It is a form of structural testing and a White Box testing technique that focuses on program variables and the paths:
    - From the point where a variable, v, is defined or assigned a value
    - To the point where that variable, v, is used
- **predicate use (p-use)** for a variable that indicates its role in a predicate.
- **A computational use (c-use)** indicates the variable's role as a part of a computation. In both cases the variable value is unchanged.
- For example, in the statement
- Y = 26 * X ,the variable X is used. Specifically it has a c-use.
- In the statement if (X= 98)
- Y =max , *X has a predicate or p-use.*

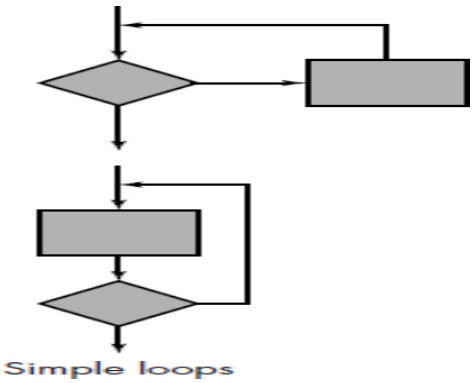
The following combinations of paths are tested:

- All p-uses
- All c-uses/some p-uses
- All p-uses/some c-uses
- All uses
- All def-use paths

Loop Testing

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

Simple loops



Simple loops

Fig.4.7 Simple loop

The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m < n.
5. n 1, n, n + 1 passes through the loop.

Nested loops

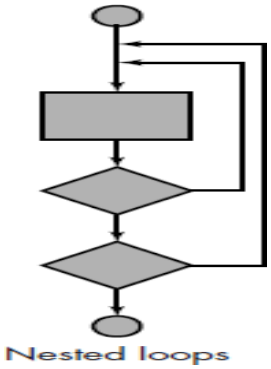


Fig.4.7. Nested loop

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

Concatenated loops

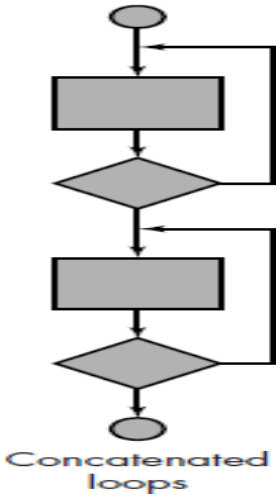


Fig.4.8. Concatenated loops

Unstructured loop

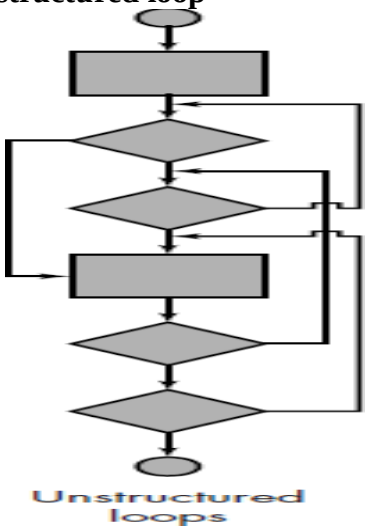


Fig.4.9. Unstructured loop

Test Case

A test case in a practical sense is a test-related item which contains the following information:

- A set of test inputs. These are data items received from an external source by the code under test. The external source can be hardware, software, or human.

- Execution conditions. These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.
- Expected outputs. These are the specified results to be produced by the code under test.

Software Testing Strategies Strategic

Approach to Software Testing

- Testing begins at module level and works outward towards the of integration entire computer based system.
- Different testing techniques are required at different points in time.
- Testing is conducted by the s/w developer and ITG(Independent Test Group) for large projects.
- Testing and Debugging are different and Debugging is essential in any testing strategy.

Verification and Validation

- Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase .
- Does the product meet its specifications?
- Are we building the product right?
- Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements
- Does the product perform as desired?
- Are we building the right product

A Software Testing Strategy

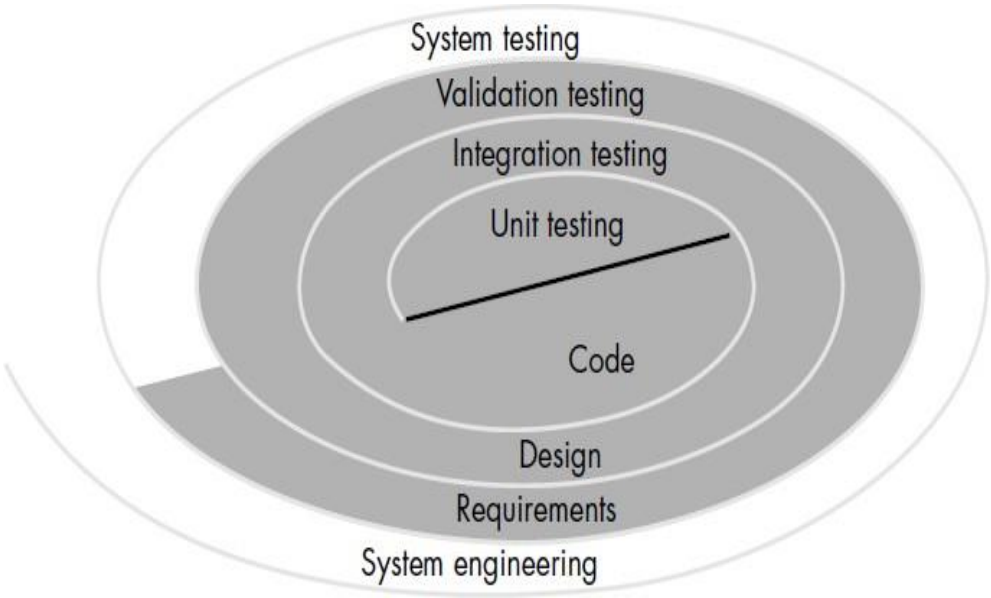


Fig.4.10. Software Testing Strategy

- A strategy for software testing may also be viewed in the context of the spiral
- Unit testing begins at the vortex of the spiral and concentrates on each unit of the software as implemented in source code.
- Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture.
- Taking another turn outward on the spiral, we encounter validation testing, where requirements established as part of software requirements analysis are validated against the software that has been constructed.
- Finally, we arrive at system testing, where the software and other system elements are tested as a whole.

Software testing steps

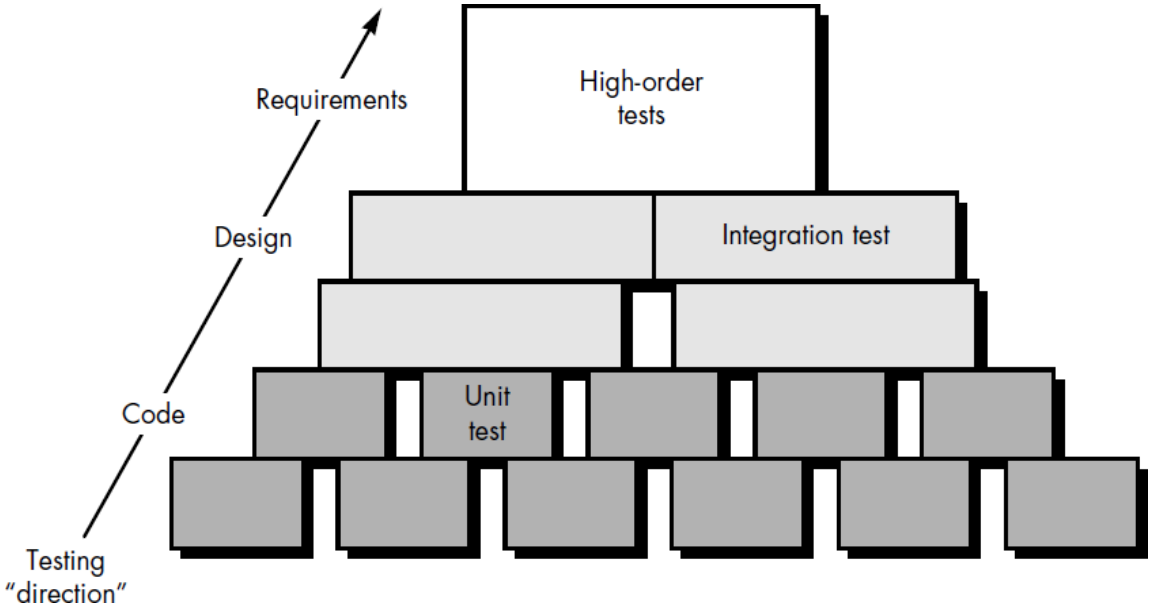


Fig.4.11. Software testing steps

Unit Testing

Unit testing focuses on the smallest element of software design viz. the module.

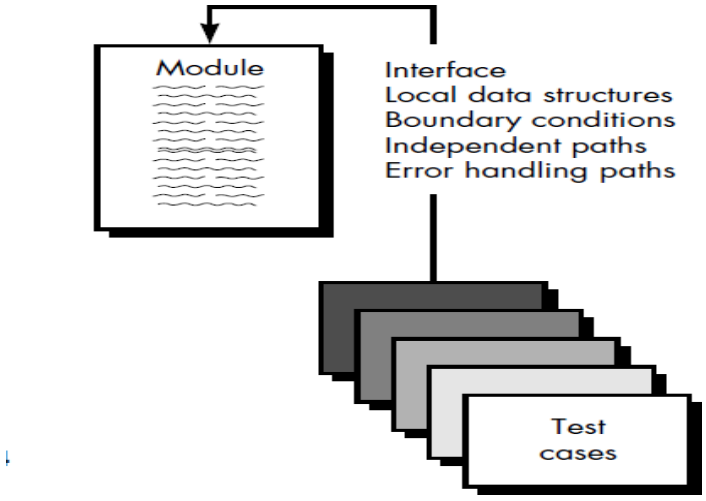


Fig.4.12. Unit Testing

- The module **interface** is tested to ensure that information properly flows into and out of

the program unit under test.

- The **local data structure** is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- **Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- **All independent paths** (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- And finally, all **error handling paths** are tested.

Unit Test Procedures

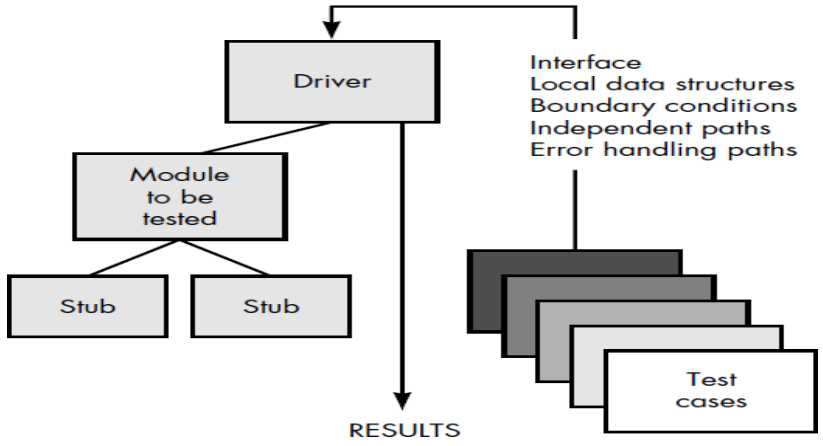


Fig.4.13. Unit Test Environment

Stubs and Drivers

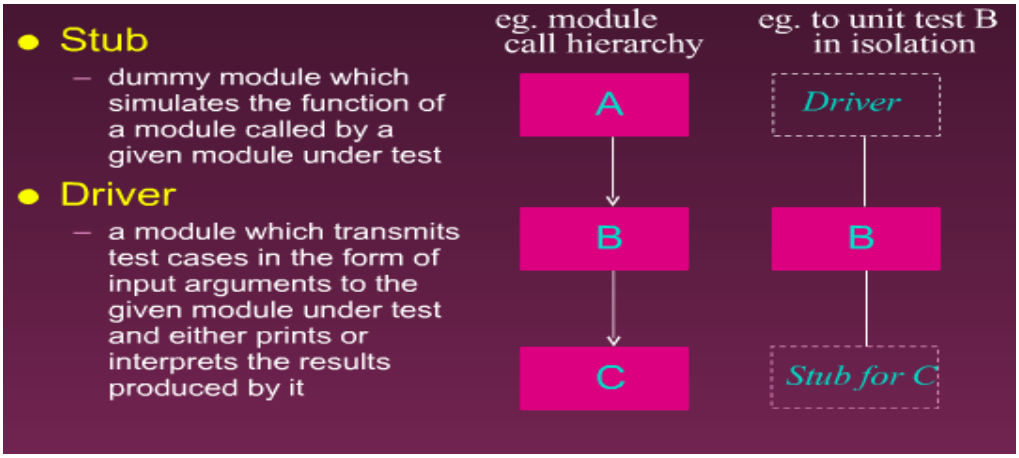


Fig.4.14. stubs and Drivers

Unit Testing: Using stubs and drivers to isolate the module under unit test

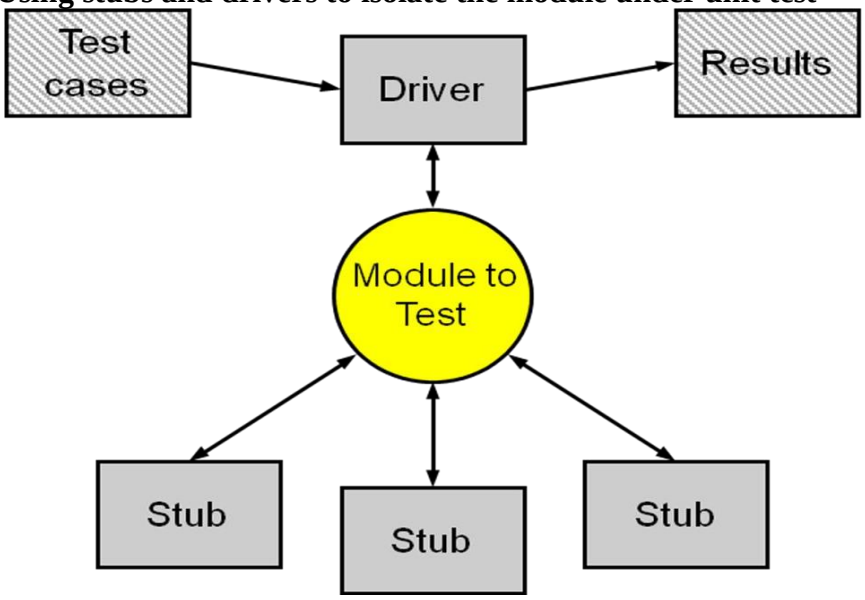


Fig.4.15. Using stubs and drivers to isolate the module under unit test

- A **driver** is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- **Stubs** serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Integration Testing

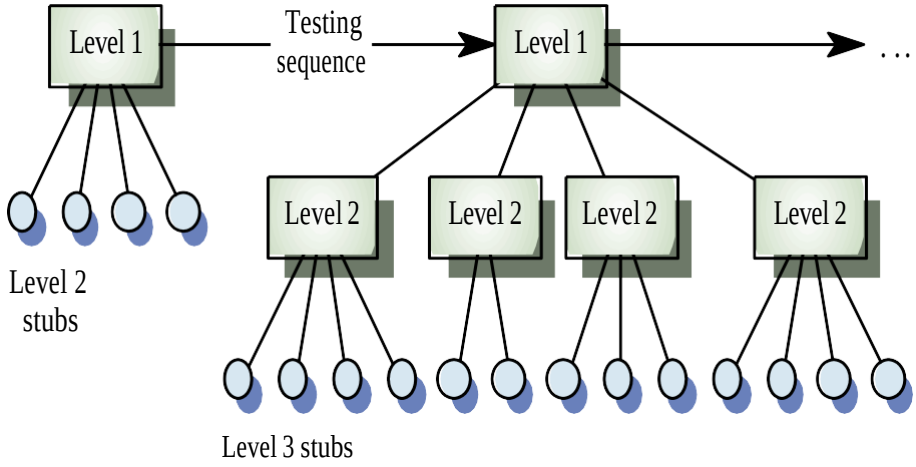
Integration testing is the phase in software testing in which individual software modules are combined and tested as a group.

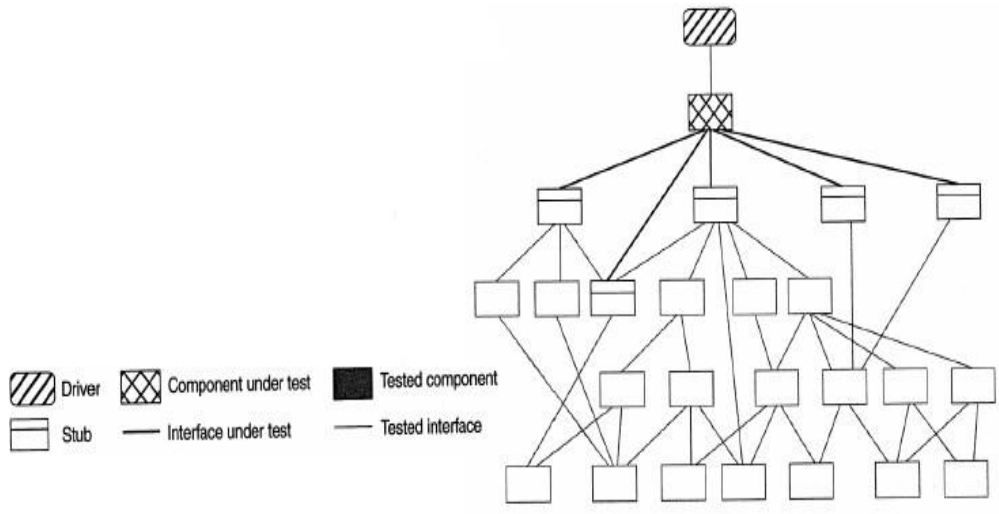
- It occurs after unit testing and before validation testing.
- Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.
- **Nonincremental integration**; that is, to construct the program using a "**big bang**" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program
- In **Incremental integration** The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Top-down Integration

- Top-down integration testing is an incremental approach to construction of program structure.
- Modules are integrated by moving downward through the control hierarchy ,beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Top-down Integration





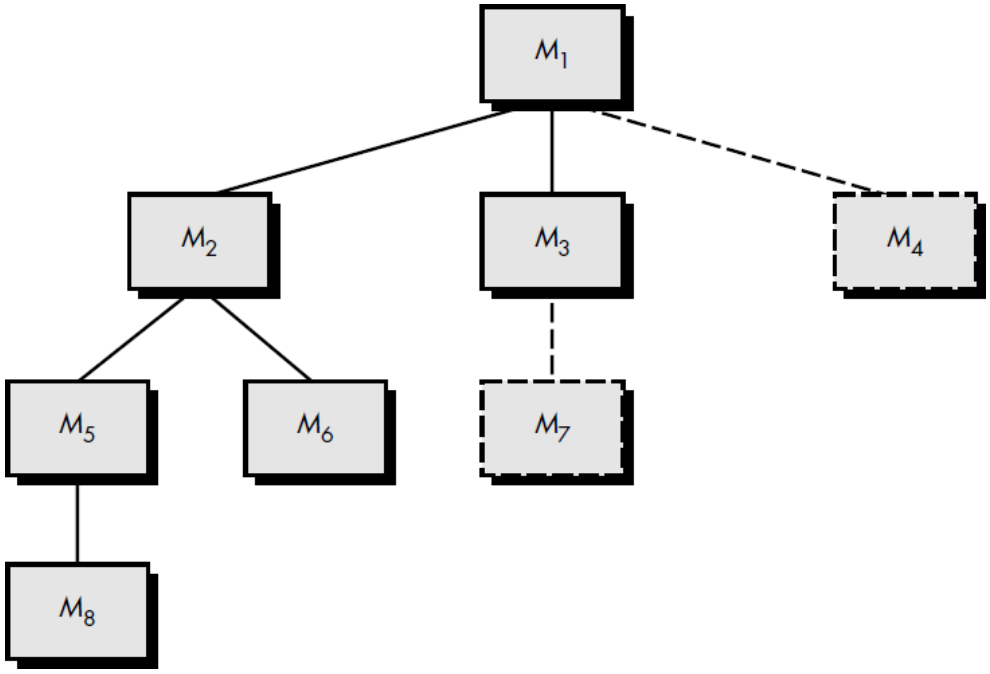


Fig.4.16. Top-down Integration

- **Depth-first integration** would integrate all components on a major control path of the structure. selecting the lefthand path, components M1, M2 , M5 would be integrated first. Next, M8 or M6 would be integrated. Then, the central and righthand control paths are built.
- **Breadth-first integration** incorporates all components directly subordinate at each level, moving across the structure horizontally. components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

The integration process is performed in a series of five steps:

- The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
- Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
- Tests are conducted as each component is integrated.
- On completion of each set of tests, another stub is replaced with the real component.
- Regression testing may be conducted to ensure that new errors have not been introduced.

Bottom-up Integration

- Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

- Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

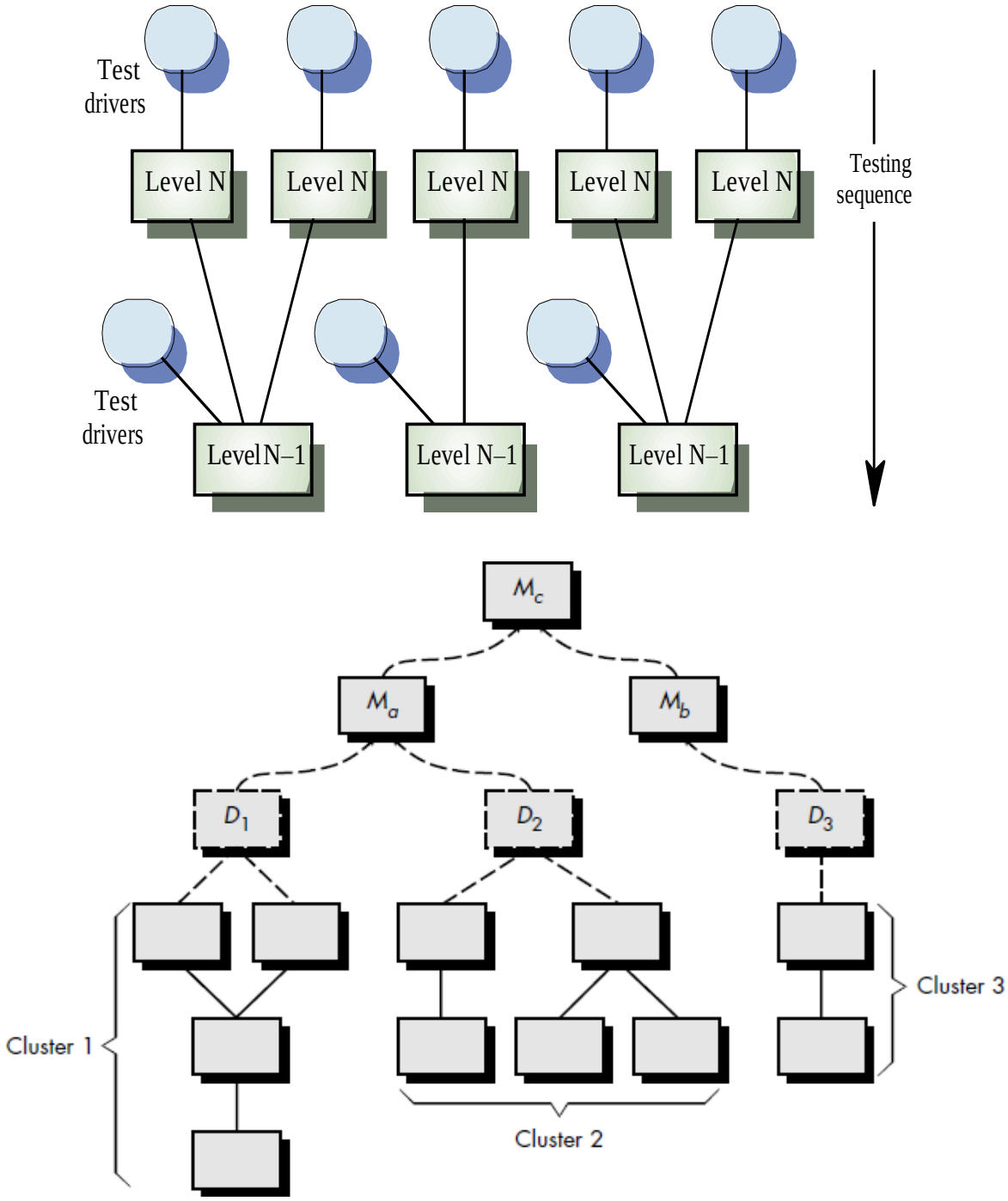


Fig.4.17. Bottom-Up Integration

- Components are combined to form clusters 1, 2, and 3.
- Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma.
- Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb.
- Both Ma and Mb will ultimately be integrated with component Mc, and so forth

Bottom-up integration strategy may be implemented with the following steps:

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

Regression Testing

- Re-executing all prior tests after a code change
  - often done by scripts, automated testing
    - used to ensure that old fixed bugs are still fixed
      - a new feature or a fix for one bug can cause a new bug or reintroduce an old bug
- especially important in evolving object-oriented systems

Smoke test

- Borrowed from hardware testing

A relatively simple check to see whether the product -smokes‖

- Check basic functionality of software
- Daily/nightly build

Software is compiled, linked and (re)tested on a daily Basis -Good‖ build if pass all smoke tests

- Software components that have been translated into code are integrated into a -build.‖ A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

- A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover -show stopper‖ errors that have the highest likelihood of throwing the software project behind schedule.
- The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

Validation Testing

- validation succeeds when software functions in a manner that can be reasonably expected by the customer.

Validation Test Criteria

- Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements.
- Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and human engineered and other requirements are met
- After each validation test case has been conducted, one of two possible conditions exist:
  - The function or performance characteristics conform to specification and are accepted or
  - a deviation from specification is uncovered and a deficiency list is created.

Configuration Review

- The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle.

Alpha and Beta Testing

- The **alpha test** is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.
- The **beta test** is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems that are encountered

during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

System Testing

- System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system

Recovery Testing

- Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic , reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits

Security Testing

- Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."
- During security testing, the tester plays the role(s) of the individual who desires to penetrate the system.

Stress Testing

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

- special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
- input data rates may be increased by an order of magnitude to determine how input functions will respond,
- test cases that require maximum memory or other resources are executed,
- test cases that may cause thrashing in a virtual operating system are designed,
- test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

Performance Testing

- The testing to evaluate the response time (speed), throughput and utilization of system to execute its required functions in comparison with different versions of the same product or a different competitive product is called Performance Testing.
- Performance testing is done to derive benchmark numbers for the system.
- Heavy load is not applied to the system
- Tuning is performed until the system under test achieves the expected levels of performance.

Load Testing

- Process of exercising the system under test by feeding it the largest tasks it can operate with.
- Constantly increasing the load on the system via automated tools to simulate real time scenario with virtual users.

Examples:

- Testing a word processor by editing a very large document.
- For Web Application load is defined in terms of concurrent users or HTTP connections.

Debugging

- Debugging is the process that results in the removal of the error.

The Debugging Process

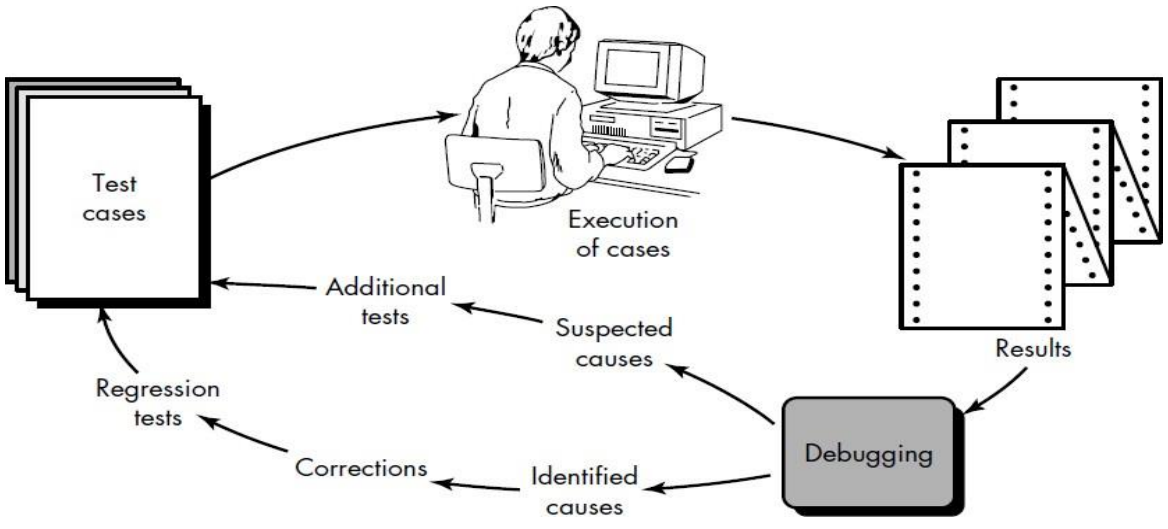


Fig.4.18. The Debugging Process

- The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.
- In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.
- The debugging process will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In

Black box Testing

- Black box approach, a tester consider the software under test to be an opaque box. There is no knowledge of its internal structure.
- The tester only has the Knowledge of what it does.
- It is often called as functional or specification based testing.

Random Testing

- Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing.
- For example, if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen

Issues

- Are the three values adequate to show that the module meets its specification when the tests are run? Should additional or fewer values be used to make the most effective use of resources?
- Are there any input values, other than those selected, more likely to reveal defects? For example, should positive integers at the beginning or end of the domain be specifically selected as inputs?
- Should any values outside the valid domain be used as test inputs? For example, should test data include floating point values, negative values, or integer values greater than 100?

Equivalence Class Partitioning

- Equivalence class partitioning results in a partitioning of the input domain of the softwareunder-test.
- Using equivalence class partitioning a test value in a particular class is equivalent to a test value of any other member of that class.
- Therefore, if one test case in a particular equivalence class reveals a defect, all the other test cases based on that class would be expected to reveal the same defect.
- We can also say that if a test case in a given equivalence class did not detect a particular type of defect, then no other test case based on that class would detect the defect

List of Conditions

1. ‘‘If an input condition for the software-under-test is specified as a range of values, select one valid equivalence class that covers the allowed range and two invalid equivalence classes, one outside each end of the range.’’

For example, suppose the specification for a module says that an input, the length of a widget in millimeters, lies in the range 1–499; then select one valid equivalence class that includes all values from 1 to 499. Select a second equivalence class that consists of all values less than 1, and a third equivalence class that consists of all values greater than 499.

2. ‘‘If an input condition for the software-under-test is specified as a number of values, then select one valid equivalence class that includes the allowed number of values and two invalid equivalence classes that are outside each end of the allowed number.’’

For example, if the specification for a real estate-related module say that a house can have one to four owners, then we select one valid equivalence class that includes all the valid number of owners, and then two invalid equivalence classes for less than one owner and more than four owners.

3.‘‘If an input condition for the software-under-test is specified as a set of valid input values, then select one valid equivalence class that contains all the members of the set and one invalid equivalence class for any value outside the set.’’

For example, if the specification for a paint module states that the colors RED, BLUE, GREEN and YELLOW are allowed as inputs,then select one valid equivalence class that includes the set RED,BLUE, GREEN and YELLOW, and one invalid equivalence class for all other inputs.

4. ‘‘If an input condition for the software-under-test is specified as a “must be” condition, select one valid equivalence class to represent the “must be” condition and one invalid class that does not include the “must be” condition.’’

For example, if the specification for a module states that the first character of a part identifier must be a letter, then select one valid equivalence class where the first character is a letter, and one invalid class where the first character is not a letter.

Boundary Value Analysis

- For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique.
- Boundary value analysis leads to a selection of test cases that exercise bounding values.

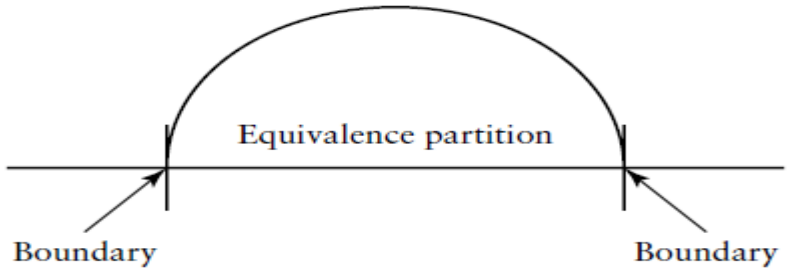


Fig.4.19. Boundaries of an Equivalence Partition

1. If an input condition for the software-under-test is specified as a range of values, develop valid test cases for the ends of the range, and invalid test cases for possibilities just above and below the ends of the range.

2. If an input condition for the software-under-test is specified as a number of values, develop valid test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum.

3.If the input or output of the software-under-test is an ordered set, such as a table or a linear list, develop tests that focus on the first and last elements of the set

The input specification for the module states that a widget identifier should consist of 3–15 alphanumeric characters of which the first two must be letters. We have three separate conditions that apply to the input: (i) it must consist of alphanumeric characters, (ii) the range for the total number of characters is between 3 and 15, and, (iii) the first two characters must be letters.

First we consider condition 1, the requirement for alphanumeric characters.This is a "must be" condition. We derive two equivalence classes.
EC1. Part name is alphanumeric, valid.
EC2. Part name is not alphanumeric, invalid.

Then we treat condition 2, the range of allowed characters 3–15.
EC3. The widget identifier has between 3 and 15 characters, valid.
EC4. The widget identifier has less than 3 characters, invalid.
EC5. The widget identifier has greater than 15 characters, invalid.

Finally we treat the "must be" case for the first two characters.
EC6. The first 2 characters are letters, valid.
EC7. The first 2 characters are not letters, invalid.

Table.4.1. Example Equivalence class reporting table

Condition	Valid equivalence classes	Invalid equivalence classes
1	EC1	EC2
2	EC3	EC4, EC5
3	EC6	EC7

A simple set of abbreviations can be used to represent the bounds groups. For example:

BLB—a value just below the lower bound
LB—the value on the lower boundary
ALB—a value just above the lower boundary
BUB—a value just below the upper bound
UB—the value on the upper bound
AUB—a value just above the upper bound

For our example module the values for the bounds groups are:
BLB—2
BUB—14
LB—3
UB—15
ALB—4
AUB—16

Table.4.2.Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module

Module name: **Insert_Widget**
Module identifier: **AP62-Mod4**
Date: **January 31, 2000**
Tester: **Michelle Jordan**

Test case identifier	Input values	Valid equivalence classes and bounds covered	Invalid equivalence classes and bounds covered
1	abc1	EC1, EC3(ALB) EC6	
2	ab1	EC1, EC3(LB), EC6	
3	abcdef123456789	EC1, EC3 (UB) EC6	
4	abcde123456789	EC1, EC3 (BUB) EC6	
5	abc*	EC3(ALB), EC6	EC2
6	ab	EC1, EC6	EC4(BLB)
7	abcdefg123456789	EC1, EC6	EC5(AUB)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6 (typical case)	

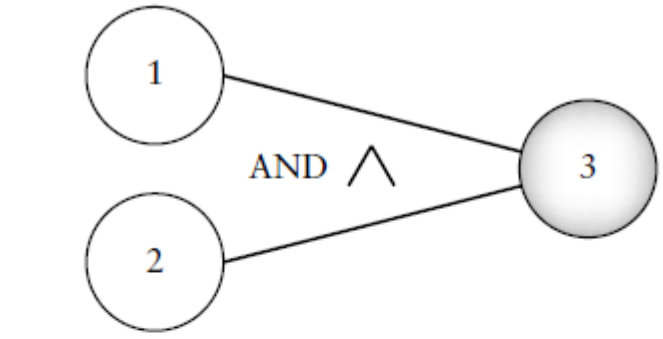
C a u s e - a n d - E f f e c t G r a p h i n g

- A major weakness with equivalence class partitioning is that it does not allow testers to combine conditions.
- Cause-and-effect graphing is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification.
- However, the specification must be transformed into a graph that resembles a digital logic circuit.

- The graph must be converted to a decision table that the tester uses to develop test cases.

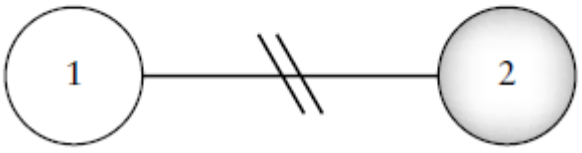
- The steps in developing test cases with a cause-and-effect graph are as follows

1. The tester must decompose the specification of a complex software component into lower-level units.
2. For each specification unit, the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. Putting together a table of causes and effects helps the tester to record the necessary details. The logical relationships between the causes and effects should be determined. It is useful to express these in the form of a set of rules.
3. From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects. Causes are placed on the left side of the graph and effects on the right. Logical relationships are expressed using standard logical operators such as AND, OR, and NOT, and are associated with arcs.
4. The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
5. The graph is then converted to a decision table.
6. The columns in the decision table are transformed into test cases.



Effect 3 occurs if both causes 1 and 2 are present.

Effect 2 occurs if cause 1 occurs.



Effect 2 occurs if cause 1 does not occur.

Fig.4.20. Sample of cause and effect graph notations

- Suppose we have a specification for a module that allows a user to perform a search for a character in an existing string.
- The specification states that the user must input the length of the string and the character to search for. If the string length is out-of-range an error message will appear.
- If the character appears in the string, its position will be reported. If the character is not in the string the message “not found” will be output.
- The input conditions, or causes are as follows:

C1: Positive integer from 1 to 80
C2: Character to search for is in string

The output conditions, or effects are:

E1: Integer out of range
E2: Position of character in string
E3: Character not found

The rules or relationships can be described as follows:

If C1 and C2, then E2.
If C1 and not C2, then E3.
If not C1, then E1.

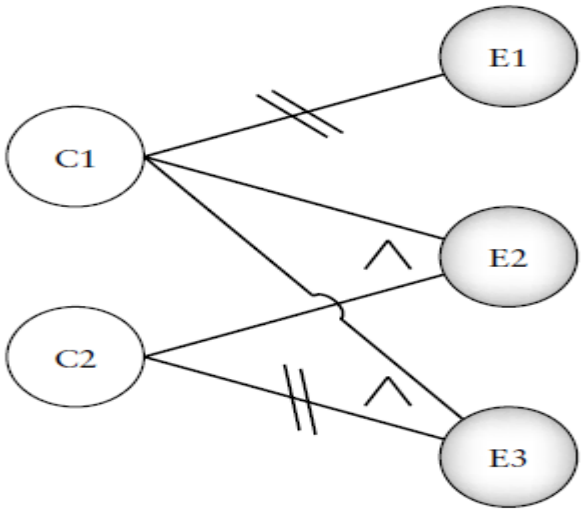


Fig.4.21. cause and effect graph for character search example

- A decision table will have a row for each cause and each effect.
- The entries are a reflection of the rules and the entities in the cause and effect graph. Entries in the table can be represented by a "1" for a cause or effect that is present, a "0" represents the absence of a cause or effect, and a "—" indicates a "don't care" value.
- A decision table for our simple example is shown in Table 4.3 where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases. The tester can use the decision table to consider combinations of inputs to generate the actual tests. In this example, three test cases are called for. If the existing string is "abcde," then possible tests are the following:

Inputs	Length	Character to search for	Outputs
T1	5	c	3
T2	5	w	Not found
T3	90		Integer out of range

Table.4.3. Decision Table

	T1	T2	T3
C1	1	1	0
C2	1	0	—
E1	0	0	1
E2	1	0	0
E3	0	1	0

State Transition Testing

- State transition testing is useful for both procedural and object-oriented development. It is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes.

- A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component.

- A finite-state machine is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

- During the specification phase a state transition graph (STG) may be generated for the system as a whole and/or specific modules. In objectoriented development the graph may be called a state chart. STG/state charts are useful models of software (object) behavior.
- STG/state charts are commonly depicted by a set of nodes (circles, ovals, rounded rectangles) which represent states. These usually will have a name or number to identify the state.
- A set of arrows between nodes indicate what inputs or events will cause a transition or change between the two linked states. Outputs/actions occurring with a state transition are also depicted on a link or arrow.

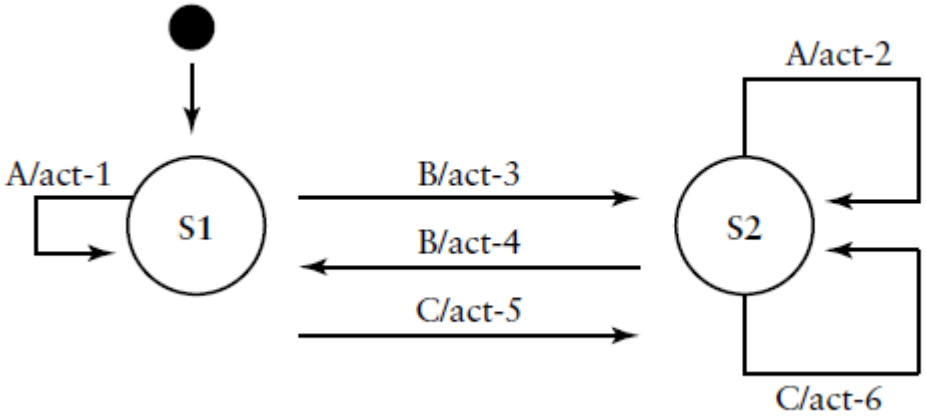


Fig.4.22. A simple state transition graph

- A simple state transition diagram is shown in Figure S1 and S2 are the two states of interest. The black dot represents a pointer to the initial state from outside the machine. Many STGs also have "error" states and "done" states, the latter to indicate a final state for the system.
- The arrows display inputs/actions that cause the state transformations in the arrow directions. For example, the transition from S1 to S2 occurs with input, or event B. Action 3 occurs as part of this state transition. This is represented by the symbol "B/act3."
- The state table lists the inputs or events that cause state transitions. For each state and each input the next state and action taken are listed. Therefore, the tester can consider each entity as a representation of a state transition.

Table.4.4. State Table

	S1	S2
Inputs		
Input A	S1 (act-1)	S2 (act-2)
Input B	S2 (act-3)	S1 (act-4)
Input C	S2 (act-5)	S2 (act-6)

E r r o r G u e s s i n g

- Designing test cases using the error guessing approach is based on the tester's/developer's past experience with code similar to the code-under test, and their intuition as to where defects may lurk in the code.
- Code similarities may extend to the structure of the code, its domain, the design approach used, its complexity, and other factors.
- The tester/developer is sometimes able to make an educated "guess" as to which types of defects may be present and design test cases to reveal them.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT-V- SOFTWARE ENGINEERING– SBS1204

SBS1204 - SOFTWARE ENGINEERING
UNIT 5

Introduction – Quality assurance – Walk through and inspections – Static analysis – Symbolic execution- Software Maintenance: Introduction – Enhancing maintainability during development- Managerial aspects of software maintenance – Configuration management – Source code metrics – Other maintenance tools and techniques

Software quality

- The degree to which a system, component, or process meets specified requirements.
- The degree to which a system, component, or process meets customer or user needs or expectations.
- Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software

Software Quality Assurance (SQA)

- A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
- A set of activities designed to evaluate the process by which the products are developed or manufactured. Contrast with: quality control.

Quality Control

- Ensure that procedures and standards are followed by the software development team.
- Quality control involves the series of inspections, reviews, and tests used throughout the software process

SQA Group Activities

- Prepare SQA plan for the project.
- Participate in the development of the project's software process description.
- Review software engineering activities to verify compliance with the defined software process.
- Audit designated software work products to verify compliance with those defined as part of the software process.
- Ensure that any deviations in software or work products are documented and handled according to a documented procedure.
- Record any evidence of noncompliance and reports them to management

Software Reviews

- Purpose is to find defects (errors) before they are passed on to another software engineering activity or released to the customer.
- Software engineers (and others) conduct formal technical reviews (FTR) for software engineers.
- Using formal technical reviews (walkthroughs or inspections) is an effective means for improving software quality.

Review Roles

• **Presenter** (designer/producer).
• **Coordinator** (not person who hires/fires).
• **Recorder**
– records events of meeting
– builds paper trail
• **Reviewers**
– maintenance oracle
– standards bearer
– user representative
– others

Formal Technical Reviews

- Involves 3 to 5 people (including reviewers)
- Advance preparation (no more than 2 hours per person) required
- Duration of review meeting should be less than 2 hours
- Focus of review is on a discrete work product
- Review leader organizes the review meeting at the producer's request.
- Reviewers ask questions that enable the producer to discover his or her own error (the product is under review not the producer)
- Producer of the work product walks the reviewers through the product
- Recorder writes down any significant issues raised during the review
- Reviewers decide to accept or reject the work product and whether to require additional reviews of product or not.

Need

• To improve quality.
• Catches 80% of all errors if done properly.
• Catches both coding errors and design errors.
• Enforce the spirit of any organization standards.
• Training

Formality and Timing

• Formal review presentations
– resemble conference presentations.
• Informal presentations
– less detailed, but equally correct.
• Early
– tend to be informal
– may not have enough information
• Later
– tend to be more formal
– Feedback may come too late to avoid rework
• Analysis is complete.
• Design is complete.
• After first compilation.
• After first test run.
• After all test runs.
• Any time you complete an activity that produce a complete work product.

Review Guidelines

• Keep it short (< 30 minutes).
• Don't schedule two in a row.
• Don't review product fragments.
• Use standards to avoid style disagreements.
• Let the coordinator run the meeting and maintain order.

Walkthroughs

- A walkthrough team should consist of four to six individuals.
- An analysis walkthrough team should include at least one representative from the team responsible for drawing up the specifications, the manager responsible for the analysis workfl ow, a client representative, a representative of the team that will perform the next workfl ow of the development (in this instance the design team), and a representative of the software quality assurance group.
- SQA group member should chair the walkthrough.
- The members of the walkthrough team should, as far as possible, be experienced senior technical staff members because they tend to find the important faults.
- The material for the walkthrough must be distributed to the participants well in advance to allow for thorough preparation.
- Each reviewer should study the material and develop two lists: a list of items the reviewer does not understand and a list of items the reviewer believes are incorrect.

Managing Walkthroughs

- The walkthrough should be chaired by the SQA representative because the SQA representative has the most to lose if the walkthrough is performed poorly and faults slip through.
- In contrast, the representative responsible for the analysis workfl ow may be eager to have the specifi cation document approved as quickly as possible to start some other task.
- The client representative may decide that any faults not detected at the review probably will show up during acceptance testing and be fi xed at that time at no cost to the client organization.
- But the SQA representative has the most at stake: The quality of the product is a direct refl ection of the professional competence of the SQA group.
- The person leading the walkthrough guides the other members of the walkthrough team through the document to uncover any faults.
- It is not the task of the team to correct faults, but merely to record them for later correction. There are four reasons for this:

- 1.A correction produced by a committee (that is, the walkthrough team) within the time constraints of the walkthrough is likely to be lower in quality than a correctionn produced by an individual trained in the necessary techniques.
- 2. A correction produced by a walkthrough team of fi ve individuals takes at least as much time as a correction produced by one person and, therefore, costs five times as much when the salaries of the fi ve participants are considered.
- 3. Not all items fl agged as faults actually are incorrect. In accordance with the dictum,"If it ain't broke, don't fi x it," it is better for faults to be analyzed methodically and corrected only if there really is a problem, rather than have a team attempt to "fi x" something that is completely correct.
- 4. There simply is not enough time in a walkthrough to both detect and correct faults. No walkthrough should last longer than 2 hours. The time should be spent detecting and recording faults, not correcting them.

- There are two ways of conducting a walkthrough.
- The first is participant driven. Participants present their lists of unclear items and items they think are incorrect. The representative of the analysis team must respond to each query, clarifying what is unclear to the reviewer and either agreeing that indeed there is a fault or explaining why the reviewer is mistaken.
- The second way of conducting a review is document driven. A person responsible for the document, either individually or as part of a team, walks the participants through that document, with the reviewers interrupting either with their prepared comments or comments triggered by the presentation. This second approach is likely to be more thorough.
- In addition, it generally leads to the detection of more faults because the majority of faults at a document-driven walkthrough are spontaneously detected by the presenter.
- Time after time, the presenter will pause in the middle of a sentence, his or her face will light up, and a
- fault, one that has lain dormant through many readings of the document, suddenly becomes obvious.
- A fruitful fi eld for research by a psychologist would be to determine why verbalization so often leads to fault detection during walkthroughs of all kinds, including requirements walkthroughs, analysis walkthroughs, design walkthroughs, plan walkthroughs, and code walkthroughs.
- Not surprisingly, the more thorough document-driven review is the technique prescribed in the IEEE Standard for Software Reviews [IEEE 1028, 1997].
- The primary role of the walkthrough leader is to elicit questions and facilitate discussion.
- A walkthrough is an interactive process; it is not supposed to be one-sided instruction by the presenter.
- It also is essential that the walkthrough not be used as a means of evaluating the participants.
- If that happens, the walkthrough degenerates into a point-scoring session and does not detect faults, no matter how well the session leader tries to run it.
- It has been suggested that the manager who is responsible for the document being reviewed should be a member of the walkthrough team.
- If this manager also is responsible for the annual evaluations of the members of the walkthrough team (and particularly of the presenter), the fault detection capabilities of the team will be compromised, because the primary motive of the presenter will be to minimize the number of faults that show up.
- To prevent this confl ict of interests, the person responsible for a given workfl ow should not also be directly responsible for evaluating any member of the walkthrough team for that workfl ow.

Inspections

- Inspections were fi rst proposed by Fagan [1976] for testing designs and code. An **inspection goes far beyond a walkthrough and has fi ve formal steps.**
- 1. An **overview of the document to be inspected (requirements, specifi cation, design,** code, or plan) is given by one of the individuals responsible for producing that document. At the end of the overview session, the document is distributed to the participants.
- 2. In the **preparation, the participants try to understand the document in detail. Lists** of fault types found in recent inspections, with the fault types ranked by frequency, are excellent aids. These lists help team members concentrate on the areas where the most faults have occurred.
- 3. To begin the inspection, one participant walks through the document with the inspection team, ensuring that every item is covered and that every branch is taken at least once. Then fault fi nding commences. As with walkthroughs, the purpose is to fi nd and document the faults, not to correct them. Within one day the leader of the inspection team (the **moderator) must produce a written report of the inspection** to ensure meticulous follow-through.
- 4. In the **rework, the individual responsible for the document resolves all faults and** problems noted in the written report.
- 5. In the **follow-up, the moderator must ensure that every issue raised has been resolved** satisfactorily, by either fi xing the document or clarifying items incorrectly fl agged as faults. All fi xes must be checked to ensure that no new faults have been introduced [Fagan, 1986]. If more than 5 percent of the material inspected has been reworked, then the team must reconvene for a 100 percent reinspection.
- The inspection should be conducted by a team of four. For example, in the case of a design inspection, the team consists of a moderator, designer, implementer, and tester.
- The moderator is both manager and leader of the inspection team. There must be a representative of the team responsible for the current workfl ow as well as a representative of the team responsible for the next workfl ow.
- The designer is a member of the team that produced thedesign, whereas the implementer is responsible, either individually or as part of a team, for translating the design into code.
- Fagan suggests that the tester be any programmer responsible for setting up test cases; it is, of course, preferable that the tester be a member of the SQA group.
- The IEEE standard recommends a team of between three and six participants [IEEE 1028, 1997]. Special roles are played by the moderator, the **reader who leads the** team through the design, and the **recorder responsible for producing a written report of** the detected faults.
- An essential component of an inspection is the checklist of potential faults. For example,the checklist for a design inspection should include items such as these: Is each item of the specifi cation document adequately and correctly addressed? For each interface, do the actual and formal arguments correspond? Have error-handling mechanisms been adequately identifi ed? Is the design compatible with the hardware resources or does it require more hardware than actually is available? Is the design compatible with the software resources; for example, does the operating system stipulated in the analysis artifacts have the functionality required by the design?
- An important component of the inspection procedure is the record of fault statistics. Faults must be recorded by severity (major or minor; an example of a major fault is one that causes premature termination or damages a database) and fault type.
- In the case of a design inspection, typical fault types include interface faults and logic faults. This information can be used in a number of useful ways:
- The number of faults in a given product can be compared with averages of faults detected at

the same stage of development in comparable products, giving management an early warning that something is amiss and allowing timely corrective action to be taken.

- If inspecting two or three code artifacts results in the discovery of a disproportionate number of faults of a particular type, management can begin checking other code artifacts and take corrective action.
- If the inspection of a particular code artifact reveals far more faults than were found in any other code artifact in the product, there is usually a strong case for redesigning that artifact from scratch and implementing the new design.
- Information regarding the number and types of faults detected at an inspection of a design artifact aids the team performing the code inspection of the implementation of that artifact at a later stage.
-

Comparison of Inspections and Walkthroughs

- Superfi cially, the difference between an inspection and a walkthrough is that the inspection team uses a checklist of queries to aid it in finding the faults. But the difference goes deeper than that.
- A walkthrough is a two-step process: preparation followed by team analysis of the document.
- An inspection is a fi ve-step process: overview, preparation, inspection, rework, and follow-up; and the procedure to be followed in each step is formalized. Examples of such formalization are the methodical categorization of faults and the use of that information in inspection of the documents of the succeeding workfl ows as well as in inspections of future products.
- The inspection process takes much longer than a walkthrough. Is inspection worth the additional time and effort? The data of Section 6.2.3 clearly indicate that inspections are a powerful, cost-effective tool to detect faults.

Software Maintenance

- Software maintenance is often considered to be an unpleasant, time consuming, expensive and unrewarding occupation - something that is carried out at the end of development only when absolutely necessary
- Modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment
- Modifying a program after it has been put into use
- Maintenance management is concerned with planning and predicting the process of change
- Configuration management is the management of products undergoing change.

Enhancing Maintainability

- Many activities during software development enhance the maintainability of software product.
  - Analysis activities
  - Standards and guidelines
  - Design activities
  - Implementation activities
  - Supporting documents
- From maintenance view point, the most important activities that occur during analysis are establishing standards and guidelines for the project and the work products to ensure uniformity of the products, setting of milestones to ensure that the work products are produced on schedule, specifying quality assurance etc.
- Software maintenance may be performed by the developing organization, by the customer,

or by a third party on behalf of the customer. In any case the customer must be given an estimate of the resources required and likely costs to be incurred in maintaining the system.

- **Standards and guidelines:** various types of standards and guidelines can be developed to enhance the maintainability of software. Standard formats for requirements documents and design specifications, structured coding conventions and standardized formats for the supporting documents like users manual etc will contribute to the understandability and hence maintainability of the software. Standards can be specified by the software quality group
- Design activities: Architectural design is concerned with developing the functional components, conceptual data structures and interconnections in software system. Detailed design is concerned with specifying algorithmic details, concrete data representations and details of the interfaces among routines and data structures.
- Implementation activities: Implementation, like design, should have the primary goal of producing software that is easy to understand and easy to modify.
- Supporting documents: Maintenance guide and test suite description are the two important supporting documents that should be prepared during the software development cycle in order to ease maintenance activities.

Managerial Aspects of Software Maintenance

- Successful software maintenance, like all software engineering activities, requires a combination of managerial skills and technical expertise. One of the most important aspects of software maintenance involves tracking and control of maintenance activities. Maintenance activity for a software product usually occurs in response to a change request filed by a user of the product.
- Change requests are usually initiated by users. A change request may entail enhancement, adaptation or error correction. A change request is first reviewed by an analyst, either closes the change request or submits to the control board the change request, the proposed fix, and an estimate of the resources required to satisfy the request.
- Change control board: The control board reviews and approves all change requests. The board may deny, recommend a modified version of change, or approve the change as submitted. The analyst provides liaison between the change control and the request initiator. Approved changes are forwarded to the maintenance programmers for action in accordance with the priority and constrains established by the change control board. The software is modified, revalidated and submitted to the change control board for approval. If the change control board approves, the master tapes and external documents are updated to reflect the changes, and the modified software is distributed to user sites as specified by the control board.
- Change Request Summaries: The status of the change requests and software maintenance activities should be summarized on a weekly or monthly basis. The summary should report emergency problems and temporary fix in effect since the last report; new change requests received and their probable dispositional ole open requests, along with the status of progress and probable closing date for each; and change requests that have been closed since the last summary report, including a description of each closed request and its disposition. In addition, a maintenance trends summary should be included in each change request summary; a trends summary graph showing the number of new requests and the total number of open requests as a function of time.
- Quality Assurance Activities: The quality assurance group should conduct audits and spots checks to determine that external documents are properly updated to reflect modifications. Quality assurance group monitors change requests, prepares change request summaries, performs regression testing of software modifications, provides configuration management, and retains and protects the physical media for software products. The group should be

represented on the change control board and should have sign-off authority for new releases of modified software products. Change control is administered by quality assurance personnel.

- Organizing maintenance programmers: Software maintenance can be performed by the development team or my members of separate organization. There are advantages and disadvantages to both approaches. Members of the development team will be intimately familiar with the product; they will understand the design philosophy of the system and why it functions as it does. Also they will take great care to design and implement the system to enhance maintainability. On the other hand they will probably be less careful in preparing the supporting documentation. Also they may be assigned to new project while retaining the responsibility for maintenance of the released product.
- Maintenance by a separate group forces more attention to standards and high quality documentation. It is also has the advantage of releasing the development team to pursue other activities. They can become highly expert on various details of the product because they devote their full attention to the product. However, a morale problem associated with maintenance programming, and rightly or wrongly a stigma is often associated with being a "maintenance programmer".
- A desirable method of organizing maintenance programming is to periodically rotate programmers between development and maintenance.

Configuration Management

- Configuration management is concerned with tracking and controlling of the work products that constitute a software product. Software tools to support configuration management include configuration management data base and version control. A configuration management data base can provide information concerning product structure, current revision number, current status and change request history for each product version.
- A version control library may be part of a configuration management data base or it may be used as a stand-alone tool.

Source-Code Metrics

- A **software metric** is a measure of some property of a piece of software or its specifications. Most of the metrics incorporate easily computed properties of the source code, such as the number of operators and operands, the complexity of the control flow graph, the number of parameters and global variables in routines and the number of levels and manner of interconnection of call graph. The approaches taken to compute a number or set of numbers that measures the complexity of the code. Thus a program with measure 10 would be more complex than a program with measure 5.

The maintenance process

- Maintenance is triggered by change requests from customers or marketing requirements Changes are normally batched and implemented in a new release of the system
- Programs sometimes need to be repaired without a complete process iteration but this is dangerous as it leads to documentation and programs getting out of step

- **The maintenance process**

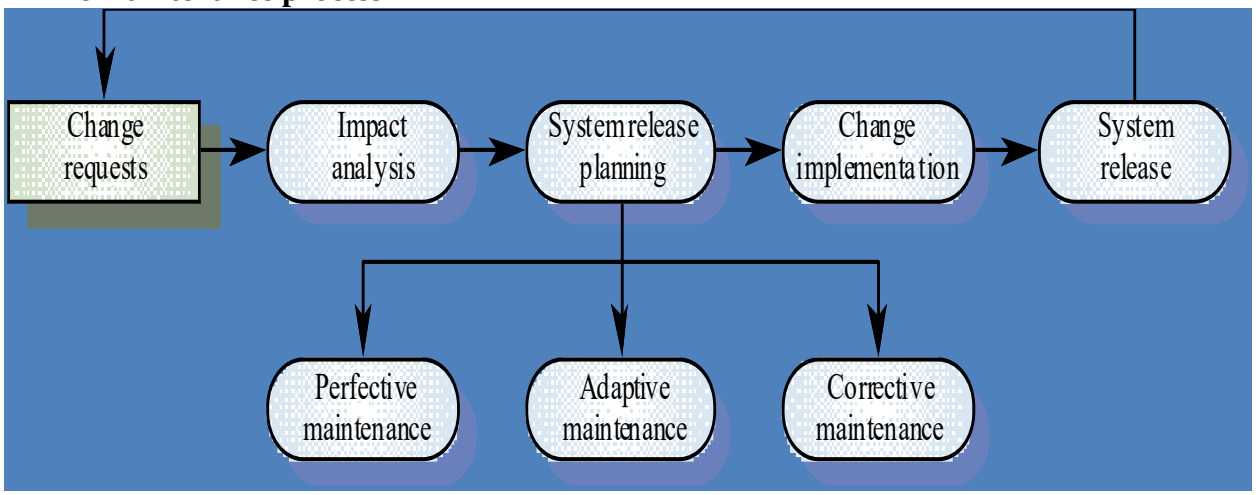


Fig.5.1. The maintenance process

Change processes

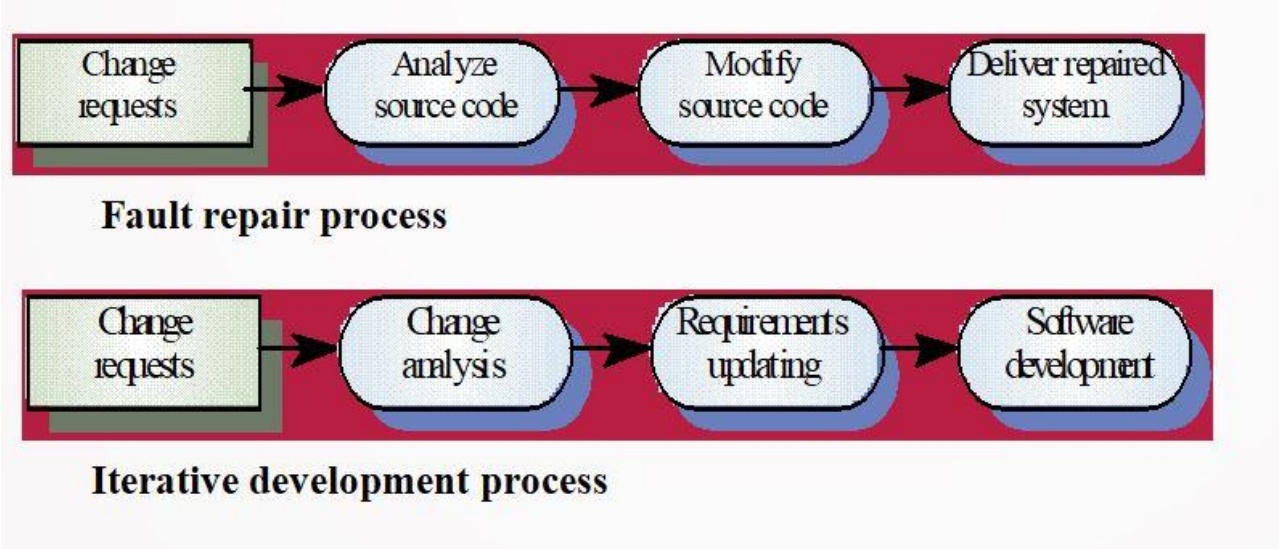


Fig.5.2. Change processes

System documentation

- Requirements document
- System architecture description
- Program design documentation
- Source code listings
- Test plans and validation reports
- System maintenance guide

The Maintenance Lifecycle

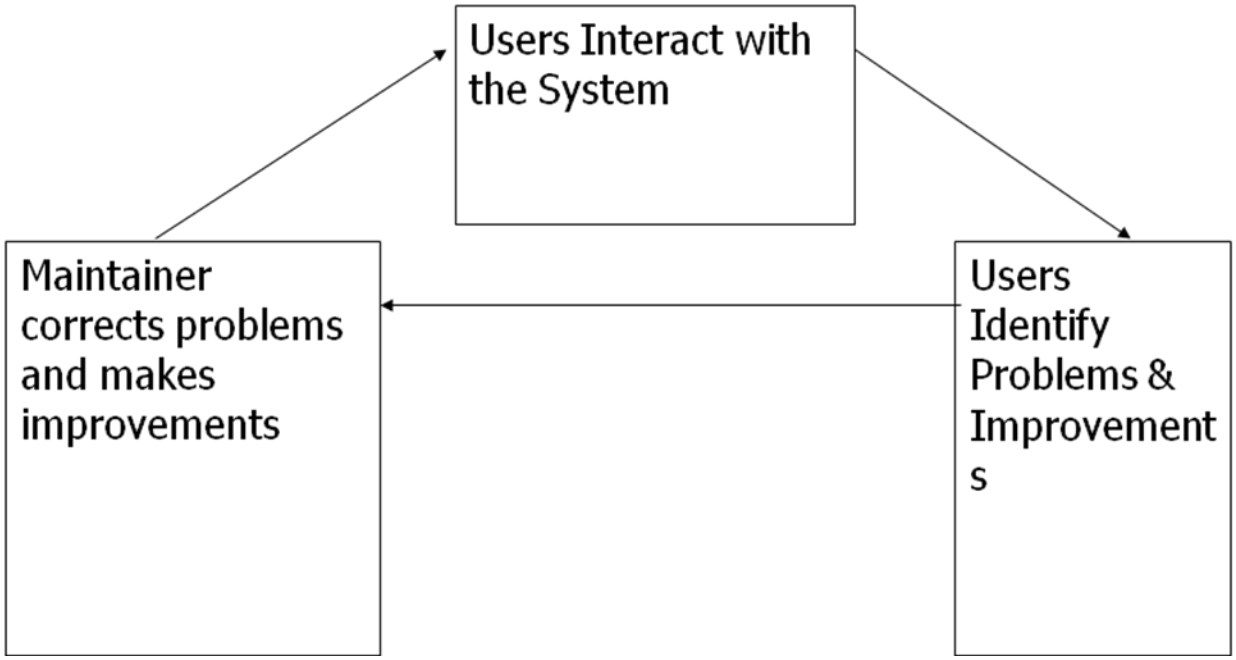


Fig.5.3. The Maintenance Life cycle

Types of Software Maintenance

- In order for a software system to remain useful in its environment it may be necessary to carry out a wide range of maintenance activities upon it. Swanson (1976) was one of the first to examine what really happens during maintenance and was able to identify three different categories of maintenance activity:

- **Corrective**

  - Changes necessitated by actual errors (defects or residual "bugs") in a system are termed corrective maintenance. These defects manifest themselves when the system does not operate as it was designed or advertised to do

- **Adaptive**

  - Any effort that is initiated as a result of changes in the environment in which a software system must operate is termed adaptive change. Adaptive change is a change driven by the need to accommodate modifications in the environment of the software system, without which the system would become increasingly less useful until it became obsolete.

- **Perfective**

  - The third widely accepted task is that of perfective maintenance. This is actually the most common type of maintenance encompassing enhancements both to the function and the efficiency of the code and includes all changes, insertions, deletions, modifications, ex-

tensions, and enhancements made to a system to meet the evolving and/or expanding needs of the user. A successful piece of software tends to be subjected to a succession of changes resulting in an increase in its requirements. This is based on the premise that as the software becomes useful, the users tend to experiment with new cases beyond the scope for which it was initially developed. Expansion in requirements can take the form of enhancement of existing system functionality or improvement in computational efficiency

- **Preventive**

- The long-term effect of corrective, adaptive and perfective change is expressed in Lehman's law of increasing entropy:

- As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.

Distribution of maintenance effort

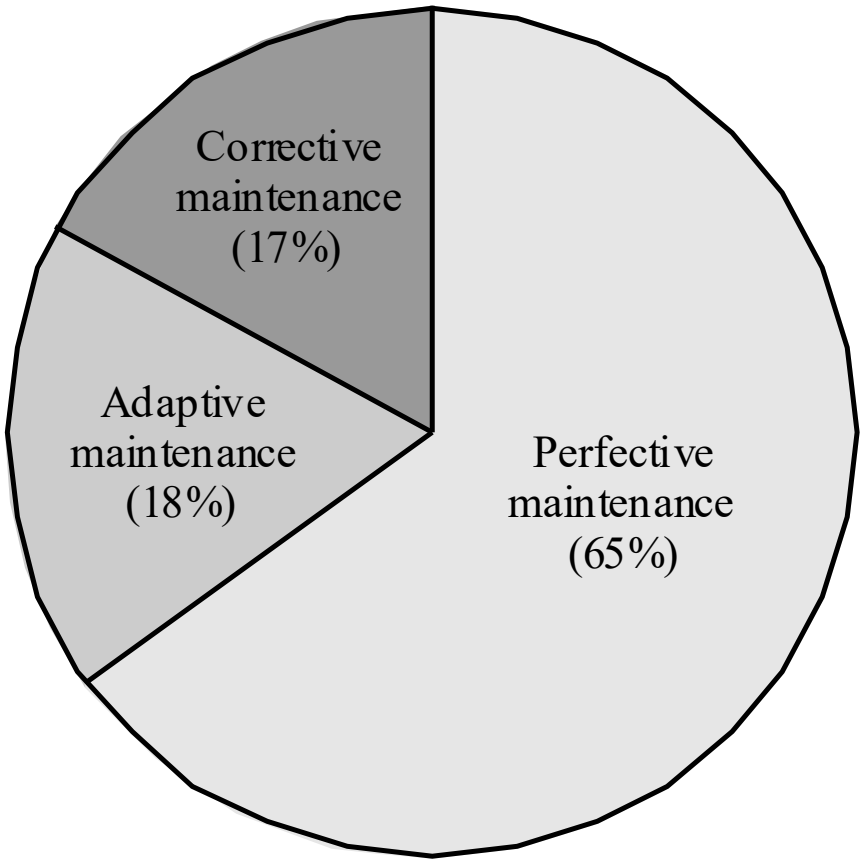


Fig.5.4. Distribution of maintenance effort