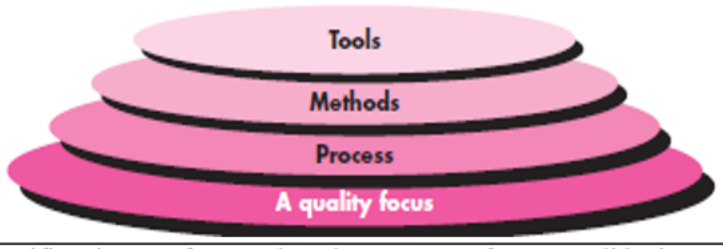
UNIT-1
SOFTWARE PROCESS AND AGILE DEVELOPMENT

Introduction to Software Engineering, Software Process, Perspective and Specialized Process Models – Introduction to Agility-Agile process-Extreme programming-XP Process.

INTRODUCTION TO SOFTWARE ENGINEERING:

Question	Answer
What is software?	Software refers to instructions (computer programs) that when executed provide desired function and performance, data structures that enable the programs to adequately manipulate information and documents that describe the operation and use of the programs.
What are the attributes of good software?	**Maintainability:**Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment. **Dependability and security:** Software dependability includes a range of characteristicsincluding reliability, security, and safety. Dependable softwareshould not cause physical or economic damage in the event ofsystem failure. Malicious users should not be able to access ordamage the system. **Efficiency:** Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc. **Acceptability:** Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.
What is software Engineering? (or) **IEEE** definition of Software Engineering?	"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 an engineering discipline that is concerned with all aspects of software production". Software Engineering is a layered technology. FIGURE 1.3 Software engineering layers 
What are the fundamental software engineering activities?	Software specification, software development, software validation, and software evolution.
What is the difference	Computer science focuses on theory and fundamentals; software

between software engineering and computer science?	engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software, and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.
What are the general issues that affect software?	1. **Heterogeneity:** software must operate on distributed systems across networks that include different types of computer and mobile devices. 2. **Business and social change**: As new technologies become available business and society change. 3. **Security and trust:** We have to make sure that malicious users cannot attack our software and that informationsecurity is maintained.
What are the costs of software engineering?	Roughly 60% of software costs are development costs; 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the Types of Software Product	1. **Generic products:** These are stand alone systems developed by organizations and sold on open market to any customer who is able to buy them. 2. **Customized products:** These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer.

SEVEN PRINCIPLES OF SOFTWARE ENGINEERING:

proposed by David Hooker

1. **The Reason It All Exists**
   - Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask questions such as: "Does this add real value to the system?"
2. **KISS (Keep It Simple, Stupid!)**
   - All design should be as simple as possible
   - This facilitates having a more easily understood and easily maintained system
3. **Maintain the Vision**
   - Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.
4. **What You Produce, Others Will Consume**
   - always specify, design, and implement knowing someone else will have to understand what you are doing.

5. **Be Open to the Future**
   - Never design yourself into a corner.
   - Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one. This could very possibly lead to the reuse of an entire system.
6. **Plan Ahead for Reuse**
   - Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.
7. **Think!**
   - Placing clear, complete thought before action almost always produces better results
   - Applying the first six principles requires intense thought, for which the potential rewards are enormous.

CHARACTERISTICS OF SOFTWARE

1. **Software is developed or engineered;** it is not manufactured in the classical sense. Software costs are concentrated in engineering. this means that software projects cannot be managed as if they were manufacturing projects.
2. **Software doesn't —wear out.**

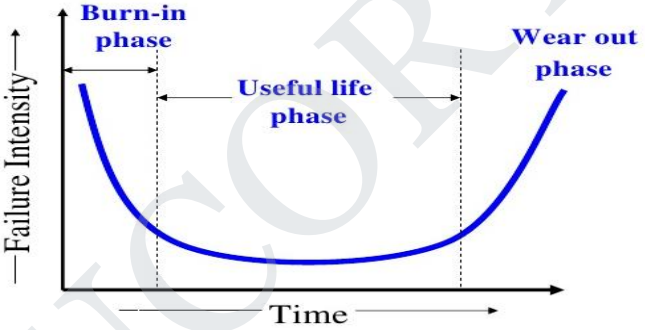


Fig: Failure curve for hardware

Figure depicts failure rate as a function of time for hardware. The relationship, often called the —bathtub curve,‖ indicates that hardware exhibits relatively high failure rates early in its life defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

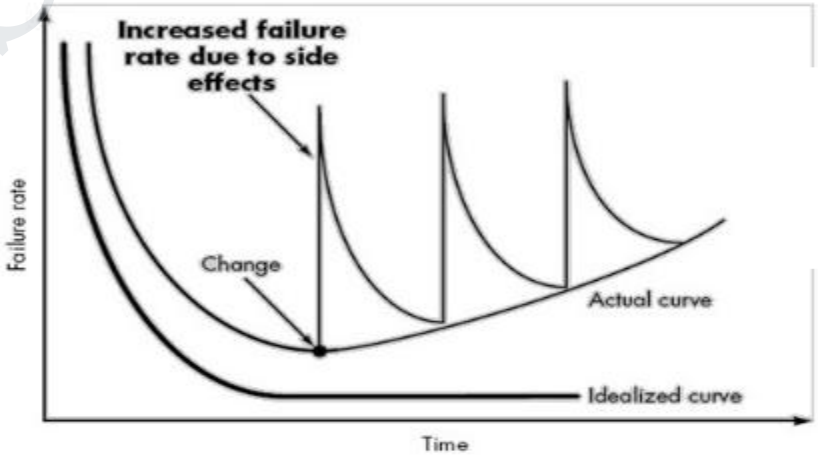


fig: Failure curves for software

PREPARED BY K.JASPIN, AP/SJIT

Considering the time curve , software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the ―actual curve‖ (Figure). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

3. **Although the industry is moving toward component-based construction, most software continues to be custom built.**

A software component should be designed and implemented so that it can be reused in many different programs.

SOFTWARE PROCESS

What is software process?	The systematic approach that is used in software engineering is sometimes called a software process. A software process is a **sequence of activities** that leads to the production of a software product.
What is a process?	**Collection of activities, actions and task**s that are performed when some work product is to be created
What is an activity?	An activity strives to achieve a **broad objective** and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. (Eg) Communication with stakeholders
What is an action?	An action encompasses a **set of tasks** that produce a major work product. (Eg) Achitectural design model
What is a task?	A task focuses on a **small well-defined objective** that produces a tangible outcome. (Eg) Conducting a unit test
What is a process framework?	A process framework establishes the foundation for a complete software engineering process. It identifies the framework activities (applicable to all software projects) and umbrella activities (applicable across the entire software process)
How to define a framework activity?	What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?

Generic Process Framework:

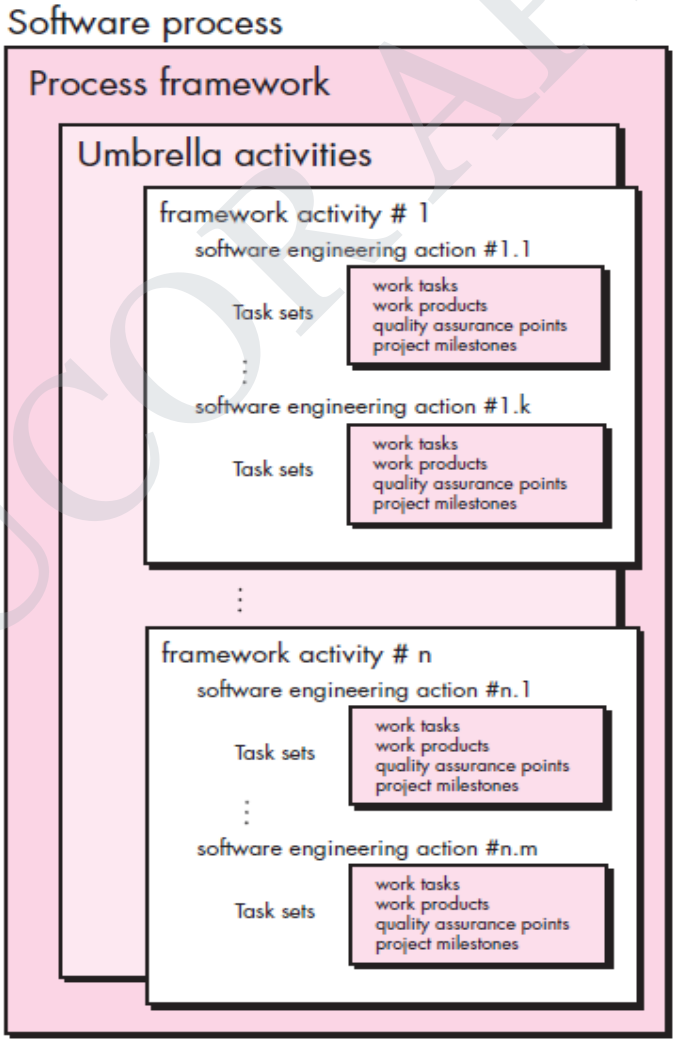
The process framework encompasses a **set of umbrella activities** that are applicable across the entire software process.

The **generic process framework** for software engineering defines **5 framework activities**:

- **Communication:** This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.
- **Planning:** This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

- **Modeling:** The activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements. The modeling activity is composed of two software engineering actions:
  1. **Analysis** encompasses a set of work tasks requirements gathering, elaboration, negotiation, specification and validation that lead to the creation of the analysis model or requirements specification.
  2. **Design** encompasses work tasks data design, architectural design, interface design and component-level design and create a design model or design specification.
- **Construction:** This activity combines code generation and the testing that is required to uncover errors in the code.
- **Deployment:** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation

FIGURE 2.1
A software process framework



Each software engineering action is represented by a number of different task sets-each a collection of software engineering work tasks, related work products, quality assurance points and project milestones. The task set that best accommodates the needs of the project and

characteristics of the team is chosen. The framework described in the generic view of software engineering is completed by a number of umbrella activities.

Typical activities under umbrella activities include:

- **Software Project Tracking and Control**-allows the software team to assess progress against the project plan and take the necessary action to maintain schedule.
- **Risk Management**-assess the risks that may effect the outcome of the project or the quality of the product.
- **Software Quality Assurance**-defines and conducts the activities required to ensure software quality.
- **Formal Technical Reviews**-assesses software engineering work products in an effort to uncover or remove errors before they are propagated to the next action or activity.
- **Measurement**-defines and collects process, project and product measures that assist the team in delivering software that meets customer needs.
- **Software Configuration Management**-manages the effects of change throughout the software process.
- **Reusability Management**-defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
- **Work Product Preparation and Production**-encompasses the activities required to create work products such as models, documents, logs, forms and lists.

All process models can be categorized within the process framework.

AGILITY

What is "Agility"?

- Agility is more than an effective (rapid and adaptive) **response to change** (team members, new technology, requirements).
- **Effective communication** in structure and attitudes among all
- Drawing the customer into the team.
- **Planning** in an uncertain world has its limits and plan must be **flexible.**
- **Organizing** a team so that it is in control of the work performed .
- Emphasize **an incremental delivery strategy** as opposed to intermediate products that gets working software to the customer as rapidly as feasible.

Yielding.. Rapid, incremental delivery of software. The development guidelines stress delivery over analysis and design although these activates are not discouraged, and active and continuous communication between developers and customers.

Agility & the Cost of Change

In software development, the **cost of change increases nonlinearly as a project progresses** (solid black curve). It is relatively **easy to accommodate a change** when a software team is gathering requirements (**early in a project**).A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project. **An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.**

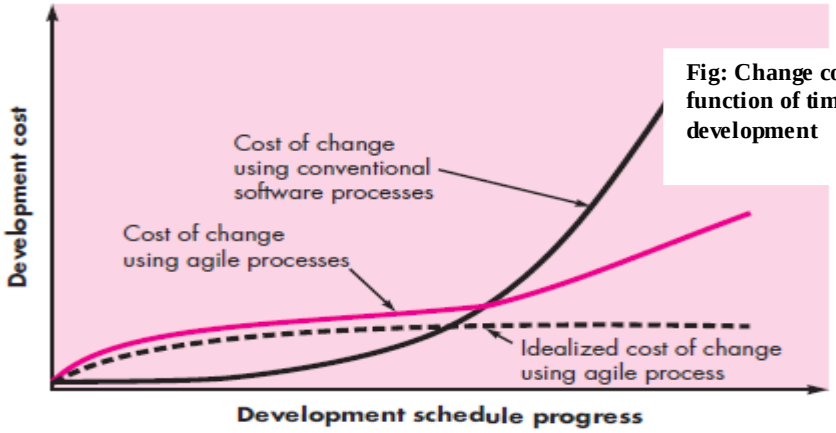


Fig: Change costs as a function of time in development

AGILE PROCESS

Any agile software process is **characterized** in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is **difficult to predict** in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are **interleaved**. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are **not as predictable** (from a planning point of view) as we might like.

12 agility principles:

It defines 12 agility principles for those who want to achieve agility:

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

Human Factors

Agile development focuses on the talents and skills of individuals. The process molds to the needs of the people and team, not the other way around. A self-organizing team is in control of the work it performs. The team makes its own commitments and defines plans to achieve them. The following **key traits** must exist among the people on an agile team and the team itself:

1. **Competence**(talent, skills, knowledge taught to all people)
2. **Common focus** (deliver a working software increment within the time promised)
3. **Collaboration** (peers and stakeholders)
4. **Decision-making ability** (freedom to control its own destiny- autonomy in project and technical issues)
5. **Fuzzy problem-solving ability** (deal with ambiguity and constant changes, today's problem may not be tomorrow's problem)
6. **Mutual trust and respect**.
7. **Self-organization** (themselves for the work done, process for its local environment, the work schedule)

EXTREME PROGRAMMING (XP)

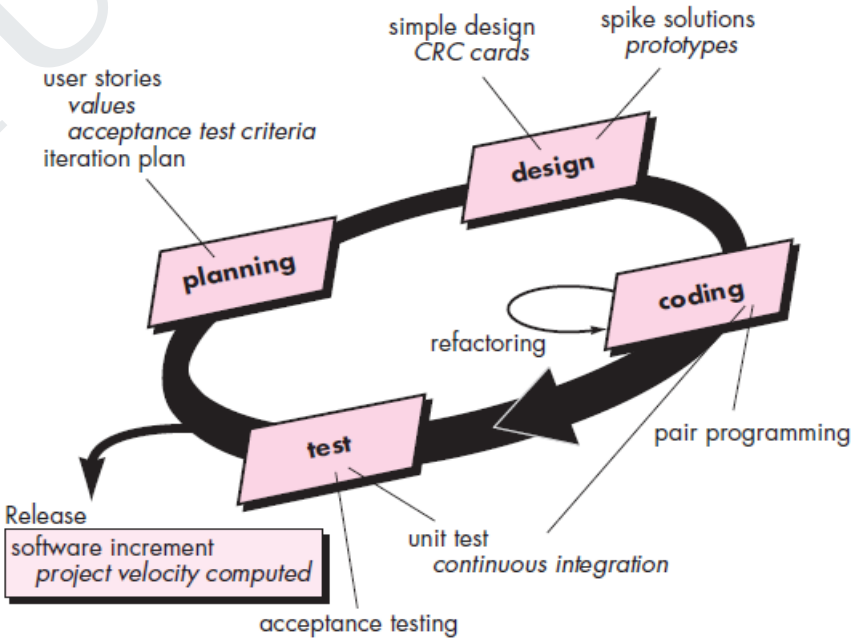
Extreme Programming (XP) is the most widely used approach to agile software development. More recently, a variant of XP, called Industrial XP (IXP) has been proposed. IXP refines XP and targets the agile process specifically for use within large organizations. "XP is the answer to the question, 'How little can we do and still build great software?'"

XP Values:

Beck defines a set of **five values** that establish a foundation for all work performed as part of XP—**communication, simplicity, feedback, courage, and respect.**

THE XP PROCESS

FIGURE 3.2
The Extreme Programming process



PREPARED BY K.JASPIN, AP/SJIT

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of **four framework activities**:

1. **Planning**
2. **Design**
3. **Coding**
4. **Testing.**

1. **Planning:**

The planning activity (also called the planning game) begins with ***listening***—a requirements gathering activity that enables the technical members of the XP team to understand requirements.

- **Listening** leads to the creation of a set of "stories" (also called user stories) that describe required output, features, and functionality for software to be built.
- Members of the XP team then assess each story and **assign a cost—measured in development weeks**—to it.
- If the story is estimated to require more than three development weeks, the customer is asked to **split** the story into smaller stories and the assignment of value and cost occurs again.
- It is important to note that new stories can be written at any time. Customers and developers work together to decide **how to group** stories into the next release (the next software increment) to be developed by the XP team.

Once a basic commitment (agreement on stories to be included, delivery date, and other project matters) is made for a release, the XP team orders the stories that will be developed in one of three ways:

(1) all stories will be implemented immediately (within a few weeks)

(2) the stories with highest value will be moved up in the schedule and implemented first,or

(3) the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes **project velocity.** Project velocity is the number of customer stories implemented during the first release.

Project velocity can then be used to:

1) **estimate delivery dates and schedule** for subsequent releases.

(2) determine whether an **overcommitment** has been made for all stories across the entire development project.

If an overcommitment occurs, the content of releases is modified or end delivery dates are changed.

2. **Design:**

- XP design follows the **KIS (keep it simple) principle**. A simple design is always preferred over a more complex representation.

- XP encourages the use of **CRC**(class-responsibility collaborator) **cards** as an effective mechanism for thinking about the software in an object-oriented context.
- If a difficult design, XP recommends the immediate creation of an operational prototype of that portion of the design called **a spike solution**, the design prototype is implemented and evaluated.

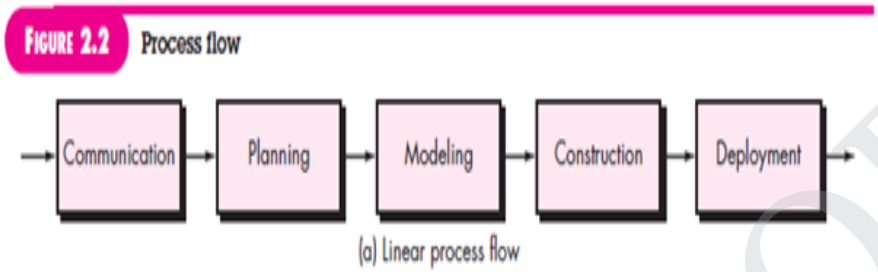
3. **Coding:**
   - After design work is done, **the team does not move to code, but rather develops a series of unit tests**.
   - **Once the unit test has been created,** the developer is better able to focus on what must be implemented to pass the test. Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.
   - A key concept during the coding activity (and one of the most talked about aspects of XP) is **pair programming**. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.
   - As pair programmers complete their work, the code they develop is integrated with the work of others.
   - This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment that helps to uncover errors early**.**
   - XP encourages **refactoring**—a construction technique that is also a method for design optimization. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure.
   - The intent of refactoring is to control these modifications by suggesting small design changes that "can radically improve the design". .
4. **Testing:**
   - The creation of **unit tests** before coding commences as a **key element** of the XP approach.
   - This encourages a **regression testing strategy** whenever code is modified.
   - As the individual unit tests are organized into a "universal testing suite", integration and validation testing of the system can occur on a daily basis.
   - **XP acceptance tests, also called customer tests**, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer**.**
   - **Acceptance tests** are derived from user stories that have been implemented as part of a software release.

PROCESS FLOW:

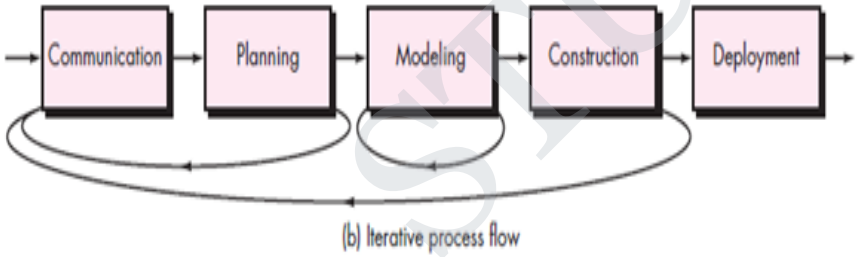
Process flow describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time. The different process flows are:

1)A linear process flow
2)An iterative process flow
3)An evolutionary process flow
4)A parallel process flow

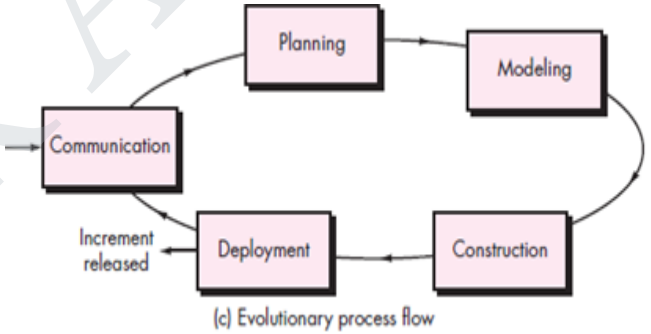
1)A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.



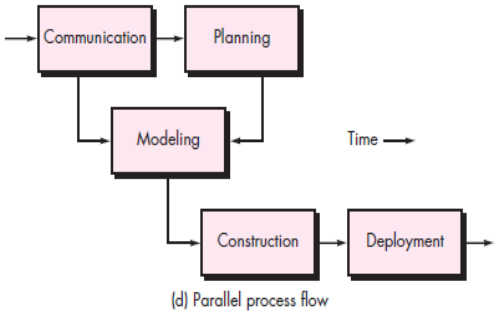
2)An iterative process flow repeats one or more of the activities before proceeding to the next.



3)An evolutionary process flow executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software.

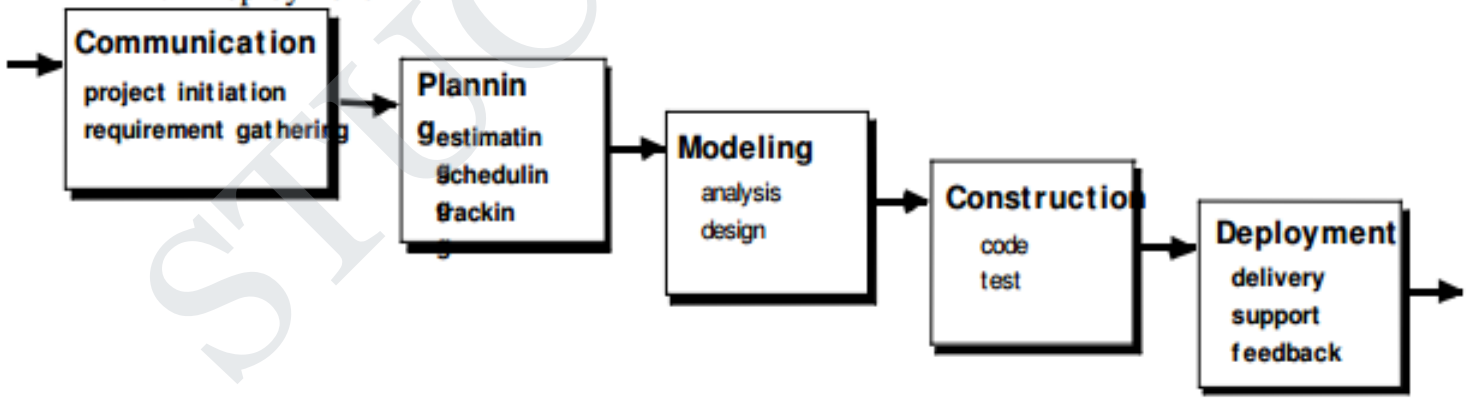


4)A parallel process flow executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).



PRESCRIPTIVE PROCESS MODELS

PROCESS MODEL	CONCEPT	ADVANTAGE	DISADVANTAGE
The waterfall model	➔ This model is also called as **Linear Sequential Model or Classic life cycle.** ➔ Waterfall model describes a **sequence of activities** in a software life cycle. ➔ This model suggests a systematic, sequential approach to software development that begins at the system level and progresses through 1)communication 2)planning 3)modeling 4)construction 5)deployment ➔ When the requirements of a problem are reasonably well understood-when work flows from communication through deployment in a reasonably linear fashion.	➔ It is the **simplest** of all models. ➔ It is a linear model ➔ **Easy to implement** ➔ **Schedules** can be **fixed easily.**	➔ It is **difficult to make modifications** in waterfall model ➔ Does not have iteration ➔ It is **not suitable** for complex, large and new projects. ➔ It may **lead to blocking states**. In blocking state situation, project team members have to wait for other members of the team to complete the dependent tasks.



THE WATERFALL MODEL

The V-model

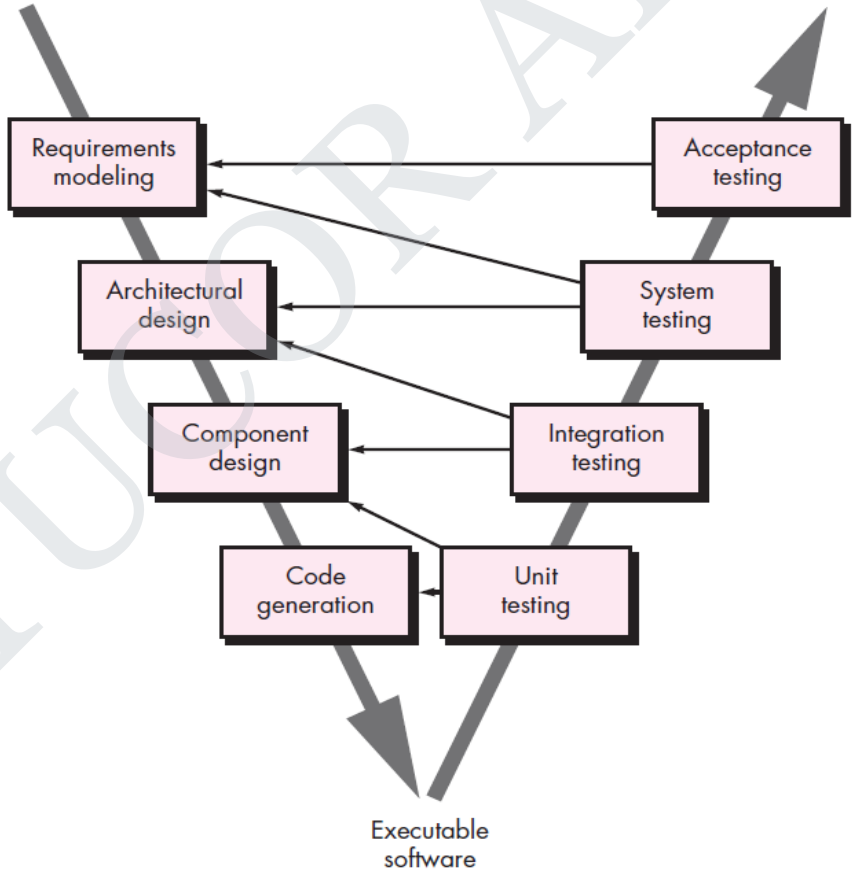
➔ A **variation in the representation of the waterfall model** is called the ***V-model***

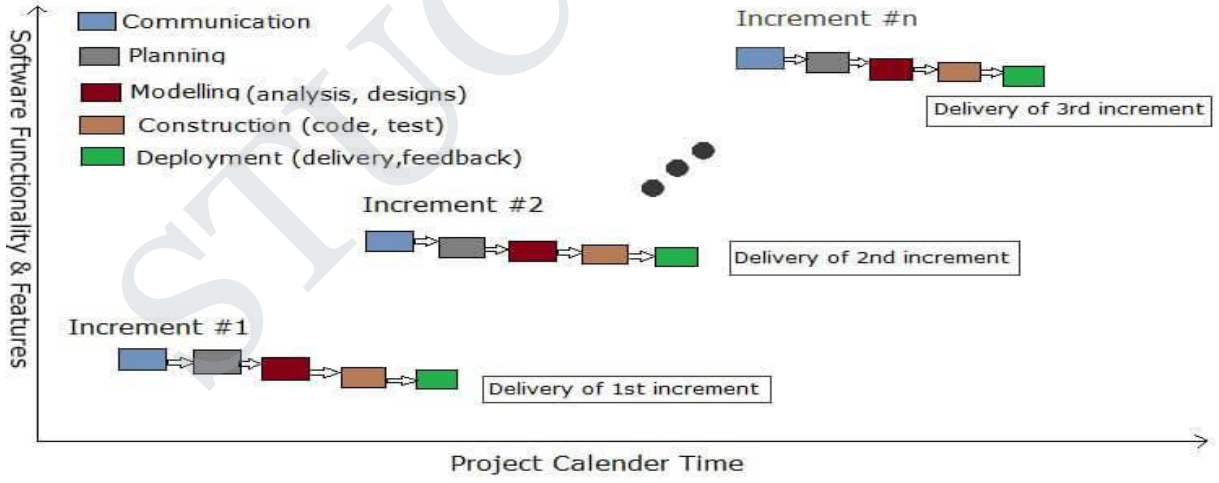
➔ V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, andearly construction activities.

➔ As a softwareteam **moves down the left side of the V**,basic problem requirements are refined into progressively more detailed and technical representations of the **problem and its solution**.

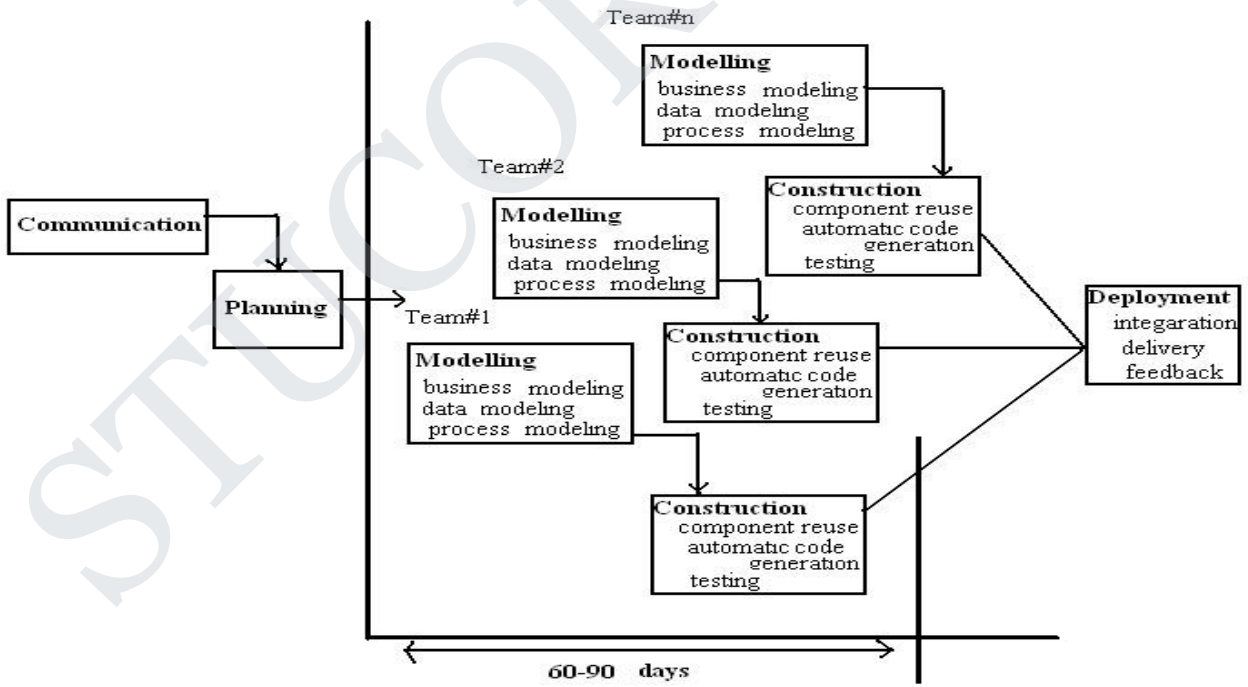
➔ Once code has been generated,the **team moves up the right side of the V**, essentially performing a **series of tests** (quality assurance actions) that validate each of the models created as the team moved down the left side.

➔ The V-model provides a way of **visualizing how verification and validation actions** are applied to earlier engineering work.



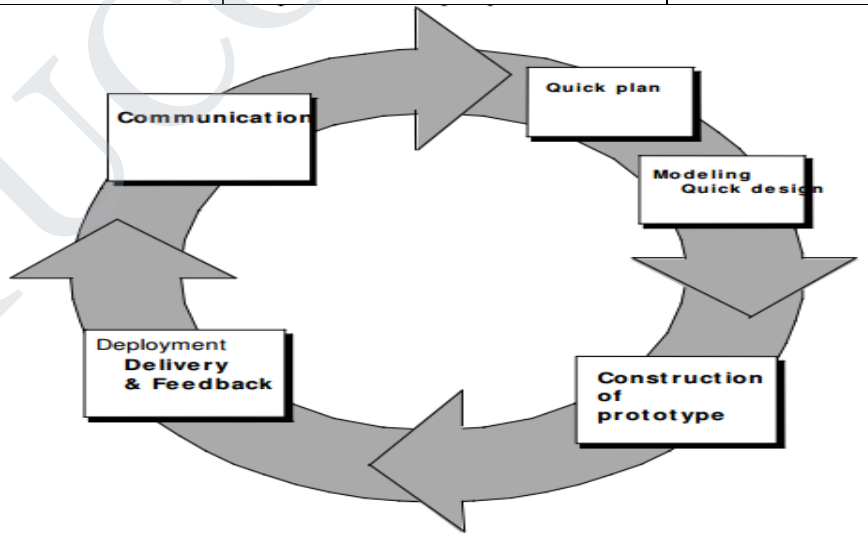
PROCESS MODEL	CONCEPT	ADVANTAGE	DISADVANTAGE
The incremental Model	➔ The incremental model combines the elements of the **waterfall model applied in an iterative fashion**. ➔ Each linear sequence produces **deliverable "increments"** of the software. ➔ When an incremental model is used, the **first increment** is often a **core product**. ➔ The core product is used and evaluated by the customer. ➔ 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. ➔ This process is **repeated until the complete product is produced.**	➔ **Suitable to manage technical risks**. ➔ Efficient when there are **less number** of people involved in the project. ➔ useful when **staffing is unavailable** for a complete implementation ➔ initial **delivery cost** is less ➔ It is **easier to test** and debug during a smaller iteration. ➔ This model is **more flexible** – less costly to change scope and requirements.	➔ **Testing cost** will be very **high**. ➔ **Requires proper planning** to distribute the work ➔ **Total cost is higher** than waterfall.
			

PROCESS MODEL	CONCEPT	ADVANTAGE	DISADVANTAGE
RAD model	➔ Rapid Application Development (RAD) is an incremental software process model that emphasizes **a short development cycle.** ➔ The RAD model is a **high-speed** adaptation of the waterfall model ➔ Rapid development is achieved by using **component-based construction** approach.	➔ create a fully functional system within a **very short time period.**	➔ RAD **requires sufficient human resources** to createthe right number of RAD teams. ➔ If developers and customers are not committed RAD **project fails.** ➔ If a system is not properly modularized, building the components for RAD will **be problematic.** High performance cannot be achieved. ➔ **Not appropriate** when technical risks are **high** or when new technology is used.



CS8494 SOFTWARE ENGINEERING

PROCESS MODEL	CONCEPT	ADVANTAGE	DISADVANTAGE
Prototyping	➔ This can be used as a stand-alone process model. ➔ This prototyping model assists the software engineer and the customer to better understand what is to be built. ➔ The prototyping begins with **communication.** ➔ A prototyping iteration **is planned quickly** in the form of **Quick design** ➔ The quick design leads to the **construction** of the Prototype. ➔ The Prototype **is deployed and then evaluated** by the customer. ➔ Feedback is used to refine requirements for the software.	➔ It **minimizes product failure**. ➔ check the function of system models before committing to a final system. ➔ This model is useful **when requirements are fuzzy.**	➔ The developer often makes implementation compromises in order to get a prototype working quickly. ➔ No consideration for **quality.** ➔ inappropriate operating system or programming language may be used ➔ inefficient algorithm may be implemented. ➔ Customer **satisfaction** is **not achieved**

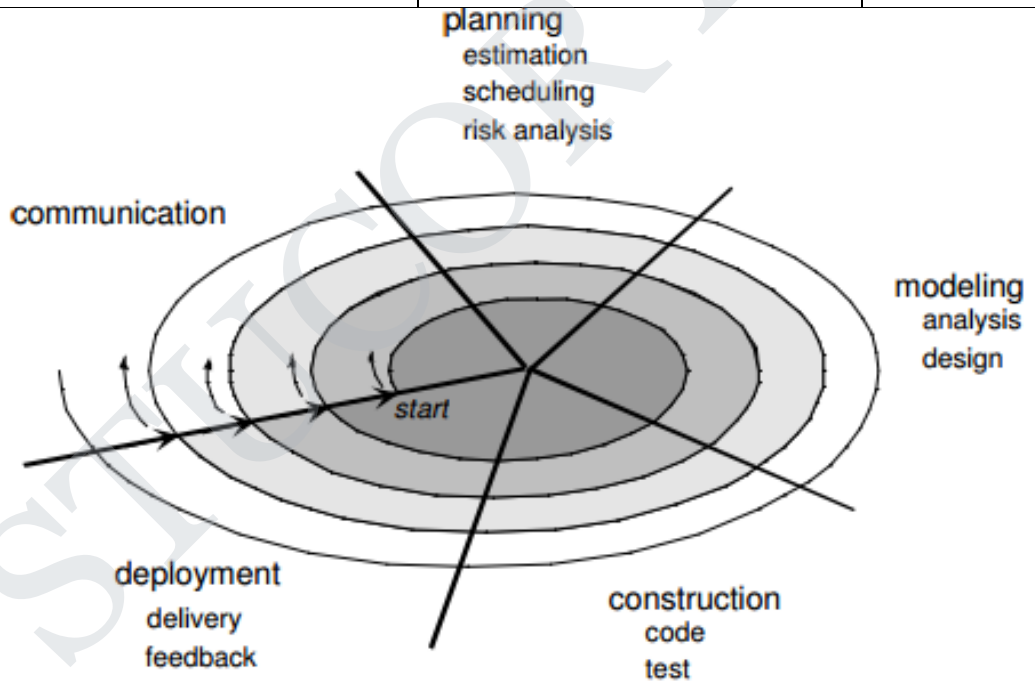


THE PROTOTYPING MODEL

PREPARED BY K.JASPIN,AP/SJIT

CS8494 SOFTWARE ENGINEERING

PROCESS MODEL	CONCEPT	ADVANTAGE	DISADVANTAGE
Spiral model	➔ It can be applied throughout the life of the computer software. ➔ It *'couples'* the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. ➔ **Anchor point milestones** -a combination of work products and conditions that are attained along the path of the spiral.	➔ **Risk rate** is **reduced**. ➔ **Reusability** of software. ➔ It can be used **when the user requirements are not clear.** ➔ Since customer involvement is there risk rate is reduced.	➔ It is only **suitable for large sized projects.** ➔ Model is **complex to use.** ➔ **Management skill** is necessary so as to analyze the risk factor.



A TYPICAL SPIRAL MODEL

PREPARED BY K.JASPIN,AP/SJIT

Concurrent Development Model

➔ This model is also called as concurrent engineering. This model has a **series of framework activities and their associated states.**

➔ Modeling activity which existed in the **inactive state** initially.

➔ Modeling activity can be in one of the states like **under development**, **waiting for modification, under revision or under review**

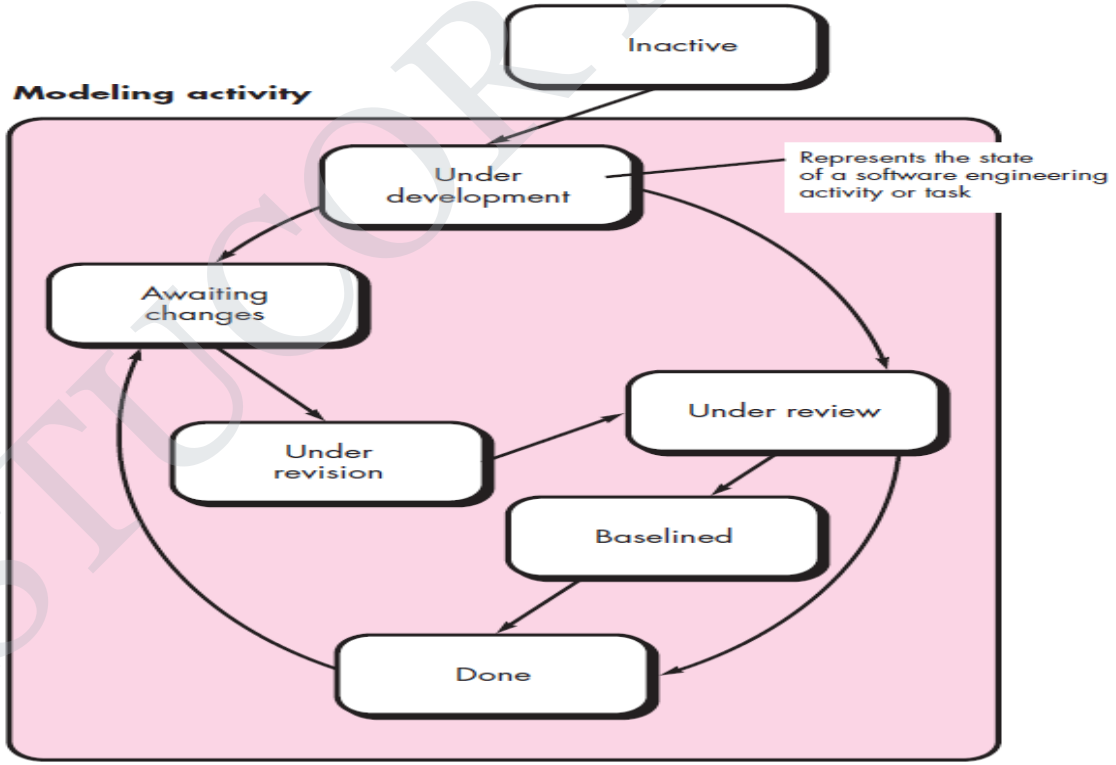
➔ The concurrent process model defines a series of events that will **trigger transitions from state to state.**

ADVANTAGE:

➔ **All type of software development** can be done using concurrent development model.

➔ This model provides **accurate picture of current state** of project.

➔ Each activity or **task can be carried out concurrently**. Hence this model is an efficient process model.



SPECIALIZED PROCESS MODELS			
PROCESS MODEL	**CONCEPT**	**ADVANTAGE**	**DISADVANTAGE**
Component-Based Development	➔ **Commercialoff-the-shelf (COTS)** software components can be used. ➔ Components should have **well-defined interfaces**. ➔ Incorporates many of the **characteristics of the spiral model.** ➔ **Evolutionary in nature.** **Steps:** ➔ Identify Candidate component ➔ Analyze Component integration issues. ➔ Design Software architecture to accommodate the components. ➔ Integrate Components into the architecture ➔ Testingis conducted to ensure proper functionality.	➔ Software **reusability.** ➔ Reusability reduces the development **cycle time** and overall **cost.**	➔ Quality of software is less because it is not under full control of developer.
Formal Methods Model	➔ Encompasses a set of activities that leads to **formal mathematical specification** of computer software ➔ Have provision to apply a **rigorous, mathematical notation**. ➔ **Ambiguity, incompleteness, and inconsistency can be discovered and corrected** more easily – not through ad hoc review, but **through the application of mathematical analysis** ➔ Offers the promise of **defect-free software** ➔ Clean room Software Engineering is a variation of the Formal Methods Model	➔ **Suitable to build the scientific models** based on mathematical techniques ➔ **Suitable for the simulation** of the some **real time systems** ➔ When there is a need to build the systems that can contribute to the **reliability and robustness (ie,critical systems)** then the formal methods models are used	➔ The development of formal models is currently quite **time-consuming and expensive** ➔ Extensive **training is required** ➔ It is difficult to use the models as a communication mechanism for **technically unsophisticated customers**

Aspect-Oriented software Development (AOSD)	➔ AOSD is also said as Aspect-Oriented Programming (AOP). ➔ AOP is a relatively new software engineering paradigm that provides a process and methodological approach for, ▪ **Defining** ▪ **Specifying** ▪ **Designing** ▪ **Constructing aspects.** ➔ Certain "concerns" – customer required properties or areas of technical interest – span the entire software architecture ➔ Example "concerns" • **Security** • **Fault Tolerance** • **Task synchronization** • **Memory Management** ➔ When concerns cut across multiple system functions, features, and information, they are often referred to as **crosscutting concerns**.

UNIFIED PROCESS MODEL

✖ It is iterative in nature.
✖ It combines the features of evolutionary model and incremental model.
✖ There are 5 phases in unified model.

1. **Inception Phase**
2. **Elaboration Phase**
3. **Construction Phase**
4. **Transition Phase**
5. **Production Phase**

1) Inception Phase

Inception phase combines communication and planning. During this phase

+ Requirements are gathered.
+ Indentify actors and their interactions
+ Develop the prototype
+ Estimate schedule and deadlines, track the schedules, and risk analysis.

2) Elaboration Phase

- It is a combination of planning and modeling.
- The initial requirements are refined and expanded.
- Requirements are made more detailed and the plan must be adjusted.
- When requirements are clear the design process starts.

3) Construction Phase

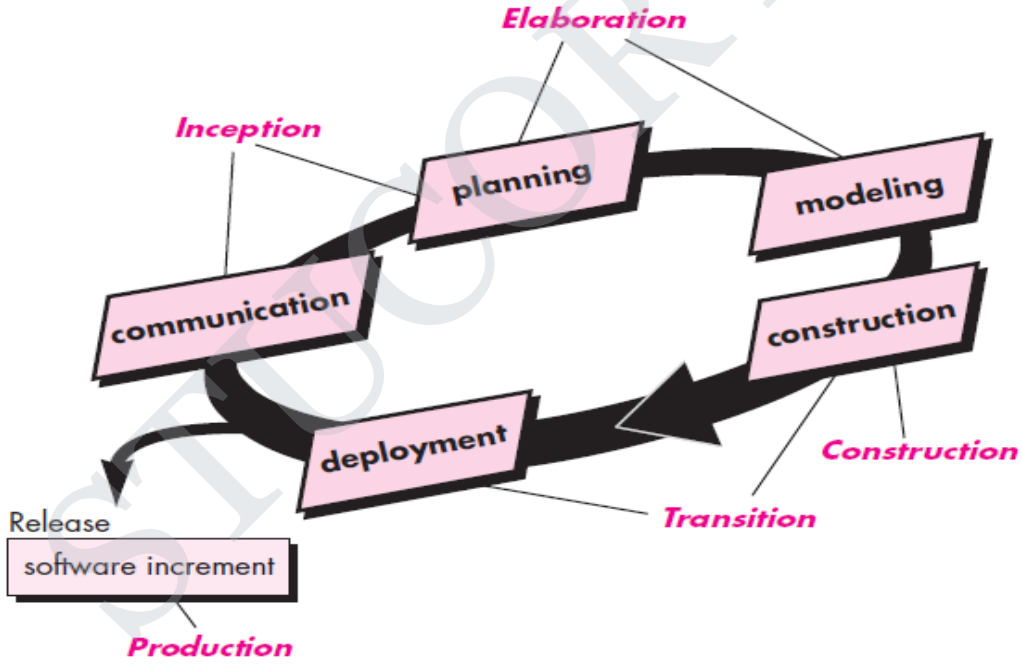
- The design is converted in to programming language and the code is tested to find errors.

4) Transition Phase

- It is the combination of construction and deployment
- The software is installed on user's machine.
- When user suggests modification the code is modified, tested and installed on the users machine.

5) Production Phase

- After the fully functional software is produced the software is released as the increment.



CS8494 SOFTWARE ENGINEERING

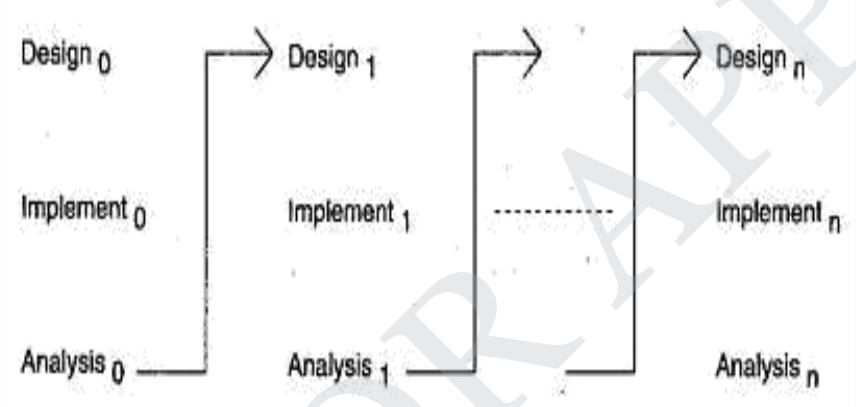
Waterfall Model	Incremental Model	RAD Model	Prototyping Model	Spiral Model	Formal Methods Model(FMM)
Requirements must be **clearly understood** and defined at the beginning only	**Requirements** are **precisely defined** and there is no confusion about the final product of the software at each increment.	**Requirements** must be **clearly understood** and defined at the beginning only	Some requirements are gathered initially but there **may be change in requirements** when the working **prototype is shown** to the customer	**Requirements** analysis and gathering can be done in iterations because **requirements get changed quite often**	**Requirements** must be **clearly understood** and defined at the beginning only
Development team having the **adequate experience** of working on the similar project is chosen to work on this type of process model	**Development team** having the **adequate experience** of working on the similar project is allowed in this type of process model	It requires heavy resources. (ie, multiple teams) **Development team** having the **adequate experience** of working on the similar project is chosen to work on this type of process model	**Development team** having the **less experience** of working on the similar project is allowed in this type of process model	**Development team** having the **less experience** of working on the similar project is allowed in this type of process model	**Development team** having the **adequate experience** of working on the similar project is chosen to work on this type of process model
If the development team has **less domain knowledge** or if it new to the technology then such a **team is allowed** for this kind of process model	The development team with **less domain knowledge** can be accommodated **due to iterative nature** of this model. The **change in technology** in the later phase **can not be tolerated**	If the development team has **less domain knowledge** or if it new to the technology then such a **team is not allowed** for this kind of process model	The **development team** has **adequate domain knowledge**. Similarly they can **adopt the new technologies** if product demands	The development team with **less domain knowledge** can be accommodated **due to iterative nature** of this model. The **change in technology** in the later phase **can not be tolerated**	If the development team has **less domain knowledge** or if it new to the technology then such a **team is not allowed** for this kind of process model

PREPARED BY K.JASPIN,AP/SJIT

Waterfall Model	Incremental Model	RAD Model	Prototyping Model	Spiral Model	Formal Methods Model(FMM)
There is **no user involvement in all the phases** of development process.	There is **no user involvement in all the phases** of development process	There is **user involvement in all the phases** of development process.	There is **user involvement in all the phases** of development process.	There is **no user involvement in all the phases** of development process.	**Limited community** makes **use** of formal methods model for their projects because this methodology is based on mathematical theorms, formal methods and automata theory.
Types of projects: When the **requirements** are reasonably **well defined** for **small systems**, the development effort suggests a purely **linear effort**then the waterfall model is chosen.	**Types of projects:** When the **requirements** are reasonably **well defined**, the development effort suggests a purely **linear effort** and when **limited set of software functionality is needed quickly** then the incremental model is chosen.	**Types of projects:** For **high speed and short time development projects** OR suitable for the projects where **technical risks are not high**. OR If there is use of **reusable components** in the project then for developing such projects this process model is.	**Types of projects:** When developer is **not sure about the efficiency of an algorithm** OR **Not sure about the adaptability of an operating system** then the prototyping model is chosen.	**Types of projects:** When the requirements are **not clearly defined**.(ie, requirements uncertainties) OR Due to iterative nature of this **risk identification and rectification** is done before they get problematic. Hence for handling **real time problems** the spiral model is chosen.	**Types of projects:** Whenever there is a need to **build the scientific models based on mathematical techniques** OR **Simulation** of the some real time systems OR When there is a need to build the systems that can contribute to the **reliability and robustness(ie,critical systems)** then the formal methods models are used.

ITERATIVE MODELS

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software for each cycle of the model.



Advantages of Iterative model:

- In iterative model we can only create a high-level design of the application before we actually begin to build the product and define the design solution for the entire product. Later on we can design and built a skeleton version of that, and then evolved the design based on what had been built.
- In iterative model we are building and improving the product step by step. Hence we can track the defects at early stages. This avoids the downward flow of the defects.
- In iterative model we can get the reliable user feedback. When presenting sketches and blueprints of the product to users for their feedback, we are effectively asking them to imagine how the product will work.
- In iterative model less time is spent on documenting and more time is given for designing.

Disadvantages of Iterative model:

- Each phase of an iteration is rigid with no overlaps
- Costly system architecture or design issues may arise because not all requirements are gathered up front for the entire lifecycle.

CMMI

The CMMI represents a process meta-model in two different ways: (1) as a continuous model and (2) as a ―staged model. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the following capability levels:

Level 0: Incomplete—the process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.
Level 1: Performed—all of the specific goals of the process area (as defined by the CMMI)have -been satisfied. Work tasks required to produce defined work products are being conducted.
Level 2: Managed—all capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are ―monitored, controlled, and reviewed; and are evaluated for adherence to the process description .
Level 3: Defined—all capability level 2 criteria have been achieved. In addition, the process is ―tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets .
Level 4: Quantitatively managed—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. ―Quantitative objectives for quality and process performance are established and used as criteria in managing the process.
Level 5: Optimized—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.
The CMMI defines each process area in terms of ―specific goals and the ―specific practices required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

UNIT II REQUIREMENTS ANALYSIS AND SPECIFICATION

Software Requirements: Functional and Non-Functional, User requirements, System requirements, Software Requirements Document – Requirement Engineering Process: Feasibility Studies, Requirements elicitation and analysis, requirements validation, requirements management-Classical analysis: Structured system Analysis, Petri Nets- Data Dictionary.

Software Requirements The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed

Requirements may be functional or non-functional.

- Functional requirements describe system services or functions
- Non-functional requirements is a constraint on the system or on the development process

Functional Requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- Describe functionality or system services
- Depend on the type of software, expected users and the type of system where the software is used
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the systemservices in detail.

Examples of functional requirements

- The user shall be able to search either all of the initial set of databases or select a subset from it.
- The system shall provide appropriate viewers for the user to read documents inthe document store.

Requirements completeness and consistency:

Complete

- They should include descriptions of all facilities required

Consistent

- There should be no conflicts or contradictions in the descriptions of the system facilities

In practice, it is impossible to produce a complete and consistent requirements document

Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.

- Process requirements may also be specified mandating a particular CASE system, programming language or development method

Non-functional requirements may be more critical than functionalrequirements. If these are not met, the system is useless.

Non-functional Classification

Product requirements

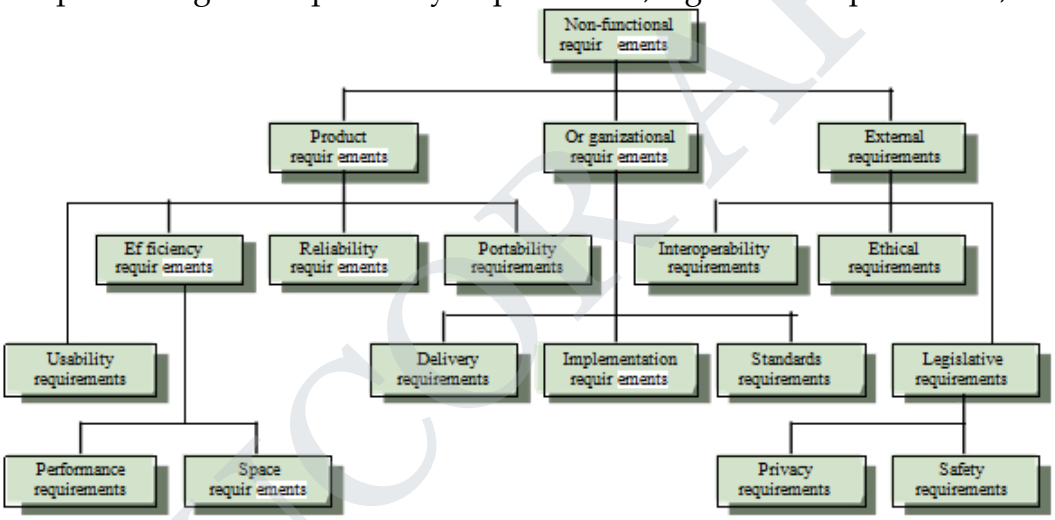
- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

Organizational requirements

- Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

External requirements

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



Examples of Non-Functional Requirements

Product Requirement

- It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set.

Organizational Requirement

- The system development process and deliverable documents shall conform to the process and deliverables in software organizations.

External Requirement

- The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Goals and Requirements

Non-functional requirements may be very difficult to state precisely and impreciserequirements may be difficult to verify.

Goal

- A general intention of the user such as ease of use Verifiable non-functional requirement
- A statement using some measure that can be objectively tested. Goals are helpful to developers as they convey the intentions of the system users.

Speed	**Processed transactions /second**
	response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of Use	Training time
	Number of help frames
Reliability	Mean time to failure
	Probability of unavailability
	Rate of failure occurrence
	Availability
Robustness	Time to restart after failure
	Percentage of events causing failure
	Probability of data corruption on failure
Probability	Percentage of target dependent statement
	Number of target systems

User Requirements

User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system.

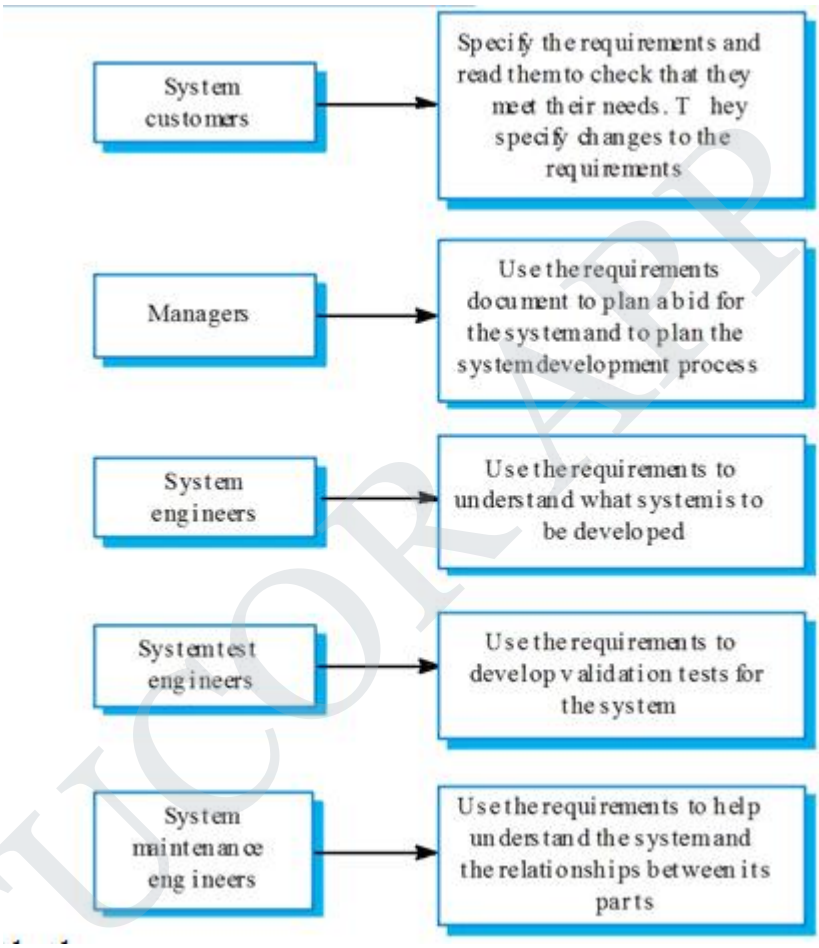
System Requirements

System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Software Requirements Document

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



IEEE Standard

- Defines a generic structure for a requirements document that must be instantiated for each specific system.
  - Introduction.
  - General description.
  - Specific requirements.
  - Appendices.
  - Index.
- Requirement Document Structure
  - Preface
  - Introduction
  - Glossary
  - User requirements definition
  - System architecture
  - System requirements specification

- System models
- System evolution
- Appendices
- Index

REQUIREMENT ENGINEERING

Requirement 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.

Guidelines Principles for Requirement Engineering

 Understand the problem before beginning 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 each and every requirements.

 Use multiple views of requirements.

 Rank the requirements and eliminate the ambiguity.

REQUIREMENT ENGINEERING PROCESS:

Inception

During inception, the requirements engineer asks a set of questions to establish…

- A basic understanding of the problem
- The people who want a solution
- The nature of the solution that is desired
- The effectiveness of preliminary communication and collaboration between the customer and the developer

Elicitation

Elicitation may be accomplished through two activities

- ✓ Collaborative requirements gathering
- ✓ Quality function deployment

Elaboration

- During elaboration, the software engineer takes the information obtained during inception and elicitation and begins to expand and refine it
- Elaboration focuses on developing a refined technical model of software functions, features, and constraints

Negotiation

- During negotiation, the software engineer reconciles the conflicts between what the customer wants and what can be achieved given limited business resources
- Requirements are ranked (i.e., prioritized) by the customers, users, and other stakeholders
- Risks associated with each requirement are identified and analyzed

Specification

A specification is the final work product produced by the requirements engineer

- It is normally in the form of a software requirements specification
- It serves as the foundation for subsequent software engineering activities

It describes the function and performance of a computer-based system and the constraints that will govern its development

Validation

- During validation, the work products produced as a result of requirements engineering are assessed for quality
- The specification is examined to ensure that
- all software requirements have been stated unambiguously
- inconsistencies, omissions, and errors have been detected and corrected
- the work products conform to the standards established for the process, the project, and the product

The formal technical review serves as the primary requirements validation mechanism

- Members include software engineers, customers, users, and other stakeholders

Requirements Management

- During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds
- Each requirement is assigned a unique identifier
- The requirements are then placed into one or more traceability tables.

FEASIBILITY STUDY

The aims of a feasibility study are to find out whether the system is worth implementing and if it can be implemented, given the existing budget and schedule.

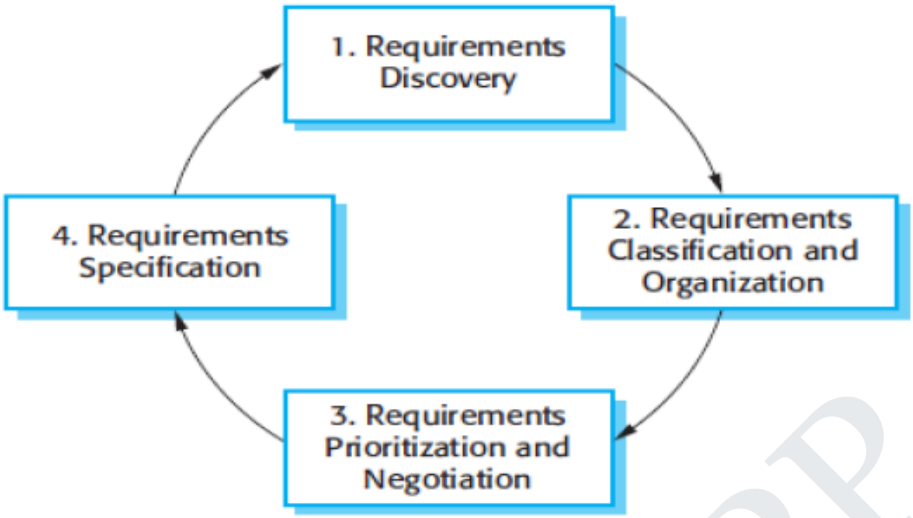
The purpose of feasibility study is not to solve the problem, but to determine whether the problem is worth solving. This helps to decide whether to proceed with the project or not.

The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it is worth carrying on with the requirements engineering and system development process.

Issues addressed by feasibility study

- Gives focus to the project and outline alternatives.
- Narrows business alternatives
- Identifies new opportunities through the investigative process.
- Identifies reasons not to proceed.
- Enhances the probability of success by addressing and mitigating factors early on that could affect the project.
- Provides quality information for decision making.
- Provides documentation that the business venture was thoroughly investigated.
- Helps in securing funding from lending institutions and other monetary sources.
- Helps to attract equity investment.
- The feasibility study is a critical step in the business assessment process. If properly conducted, it may be the best investment you ever made Carrying out a feasibility study involves information assessment, information collection and report writing.

REQUIREMENTS ELICITATION AND ANALYSIS



The process activities are:

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

Requirements classification and organization: This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.

Requirements prioritization and negotiation: Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.

Requirements specification: The requirements are documented and input into the next round of the spiral.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

- Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
- Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
- Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
- Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
- The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Requirements discovery

Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information.

Sources of information during the requirements discovery phase include documentation, system stakeholders and specifications of similar systems.

Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system.

For example, system stakeholders for the mental healthcare patient information system include:

- Patients whose information is recorded in the system.
- Doctors who are responsible for assessing and treating patients.
- Nurses who coordinate the consultations with doctors and administer some treatments.
- Medical receptionists who manage patients' appointments.
- IT staff who are responsible for installing and maintaining the system.
- A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
- Healthcare managers who obtain management information from the system.
- Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Interviewing

The requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions.

Interviews may be of two types:

- Closed interviews, where the stakeholder answers a pre-defined set of questions.
- Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

It can be difficult to elicit domain knowledge through interviews for two reasons:

All application specialists use terminology and jargon that are specific to a domain. It is impossible for them to discuss domain requirements without using this terminology. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.

Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning.

Effective interviewers have two characteristics:

1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.

2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Scenarios

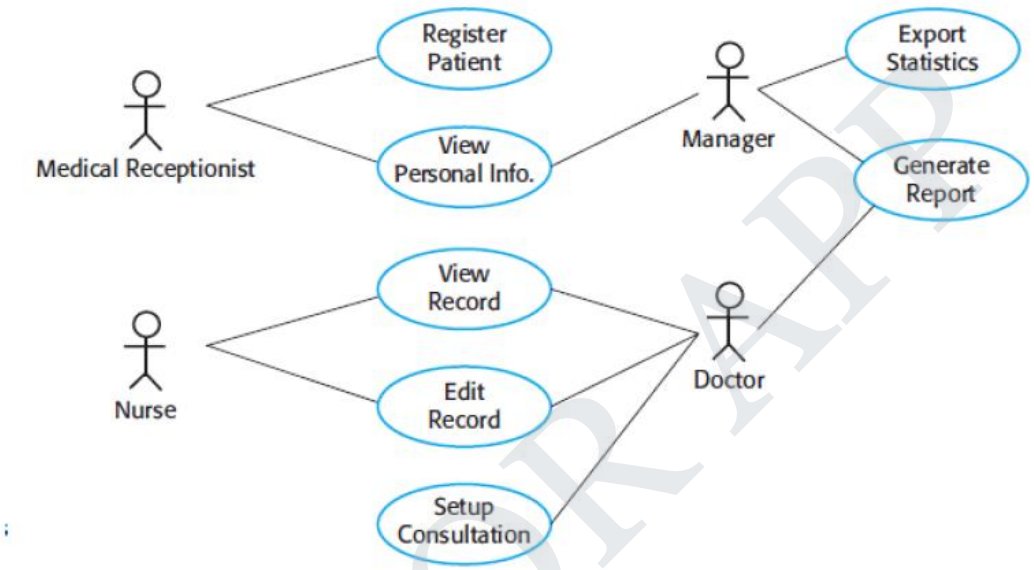
A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction.

A scenario may include:

- A description of what the system and users expects when the scenario starts.
- A description of the normal flow of events in the scenario.
- A description of what can go wrong and how this is handled.
- Information about other activities that might be going on at the same time.

A description of the system state when the scenario finishes.

Use cases



Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements.

Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail.

Ethnography

Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes. An analyst immerses himself or herself in the working environment where the system will be used. The day-to-day work is observed and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

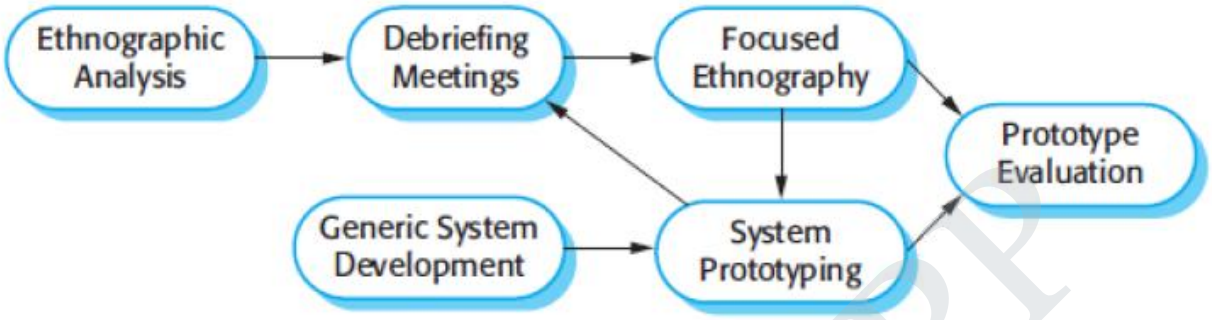
Ethnography is particularly effective for discovering two types of requirements:

Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work.

Requirements that are derived from cooperation and awareness of other people's activities.

Ethnography can be combined with prototyping

The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer.



REQUIREMENTS VALIDATION

Requirements validation is the process of checking that requirements actually define the system that the customer really wants. The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

Validity checks A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.

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

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

Realism checks Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.

Verifiability To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

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

1. Requirements reviews: The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

2. Prototyping: In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

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

REQUIREMENTS MANAGEMENT

Once a system has been installed and is regularly used, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on their business processes and the way that work is done.

There are several reasons why change is inevitable:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery; new features may have to be added for user support if the system is to meet its goals.
3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements Management Planning

Planning is an essential first stage in the requirements management process.

During the requirements management stage, the following is decided

Requirements identification: Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.

Change management process: This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.

Traceability policies: These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.

Tool support Requirements management: involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

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

Requirements storage: The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

Change management: The process of change management is simplified if active tool support is available.

Traceability management: Tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

Requirements change management

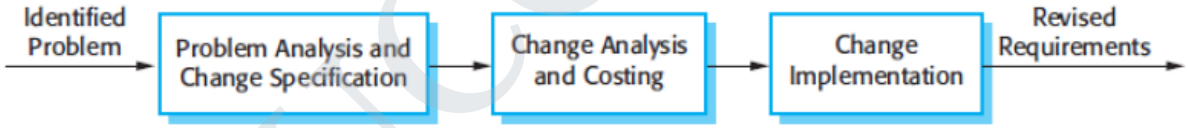
Requirements change management should be applied to all proposed changes to a system's requirements after the requirements document has been approved.

There are three principal stages to a change management process:

1. Problem analysis and change specification: The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. Change analysis and costing: The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

3. Change implementation: The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.



Classical Analysis

Structured analysis views a system from the perspective of the data flowing through it. The function of the system is described by processes that transform the data flows. Structured analysis takes advantage of information hiding through successive decomposition (or top down) analysis. This allows attention to be focused on pertinent details and avoids confusion from looking at irrelevant details. As the level of detail increases, the breadth of information is reduced. The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements. The goal of the classical analysis workflow is to produce a detailed specifications document based on the identified requirements.

Specifications doc is significant because:

- this is contract between developer and client regarding what the system will do
- if developer and client are different organizations, it is a legal contract
- it needs to address both functional and non-functional requirements of the system

- it is blueprint that designer will use for design then programmer to implement

The specifications document thus must be *detailed*, *unambiguous*, and *complete* model of the system.

Function-oriented

- top-down decomposition of business process
- each decomposition results in set of two or more simpler sub processes
- recursively decompose until function becomes trivial or easily understood/expressed
- results in tree structure that resembles organizational chart

Process-oriented

- first apply functional decomposition to identify processes/subprocesses
- use arrows to indicate directed activity
- control flow from process to process is a directed activity
- data flow is a directed activity (e.g. from database to process or vice versa)
- control flow typically expressed using *Flowcharts*
- data flow typically expressed using *Data Flow Diagrams (DFD)*
- functional decomposition and flowcharts dominated for technical/scientific systems development

Data-oriented

- Identify *data entities* in the system
- "something that has separate and distinct existence in the world of the users and is of interest to the users in that they need to record data about it".
- Entities are identified by certain *nouns* in a system description.
- Identify *entity types* by grouping together similar entities. This is an important form of abstraction.
- Each entity is an *occurrence of* its type.
- Then determine what *attributes* those entity types have

- Attributes are the relevant properties that an entity has
- All entities of a type have the same attributes
- Different entities of the same type will have different values for some attributes
- Attribute values will record an entity's *state*
- An attribute whose value uniquely identifies an entity may be selected as a *key*
- If no single attribute has unique value, a combination of attributes may be used.

- Then determine what *associations*, or interactions, those entities have with other entities.

- one entity can take an action with another, or may play a role for another
- Associations are identified by certain *verbs* or verb phrases
- associations are also known as *relationships*
- the *entity-relationship diagram* (ERD) was developed for modeling

- consider some examples: car, bookstore, library, table factory

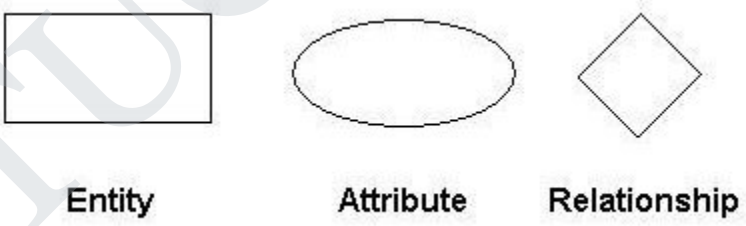
More on Relationships

- relationships are also known as associations
- each relationship is between two entity types
  e.g. in the library example, the phrase "a patron borrows books" describes a relationship "borrows" between two entity types, "patron" and "book"
- a relationship is normally *directed*, e.g. in the library example, when a patron borrows a book, the "borrows" relationship is directed from the patron to the book
- each end of the association is also characterized by its *cardinality*
- Cardinality is number of occurrences of each entity type involved in an association e.g. in the library example, one patron borrows one or more books. Cardinality on the patron end is 1 and on the book end is many (usually denoted by M or N or * or a triangle)

Entity-Relationship Diagrams

- visual modeling technique, a.k.a. "ERD" or "ER diagram"
- key concepts are described above: entities, attributes, relationships, cardinality
- ERDs describe *static* system views; DFDs show system *dynamics*
- ERDs and DFDs complement each other and Structured Analysis uses both
- database layouts and structures can be designed from ERDs
- as with DFDs, there are several different graphical notations
- most common notation is from Chen (ERD originator) or derivative:

  Typically boxes for entities, ovals for attributes, diamonds for named relationships, lines to connect attributes to entities, arrows to connect entities and relationships



Formal Classical Specification Techniques

State Transition Diagrams
Frequently used to model event sequences, GUIs, and much more. Similar to Turing Machines as computational model.

Petri Nets
Graphical notation used to model concurrent processing.
Z specification language

- formal specification notation based on set theory and first order predicate logic.
- named after German mathematician Zermelo
- pronounced "zed", European pronunciation of the letter Z

- used mostly in Europe
- methodology-independent
- produces precise unambiguous specifications
- some mathematical skills required
- fundamental entity is the *schema*
  - data schema consists of: name, subcomponents, invariants
  - operation schema consists of: name, parameters, pre- and post-conditions
- non-standard object-oriented versions exist

Z Example
Sign on an escalator:

SHOES MUST BE WORN.
DOGS MUST BE CARRIED.

Any ambiguities there?!

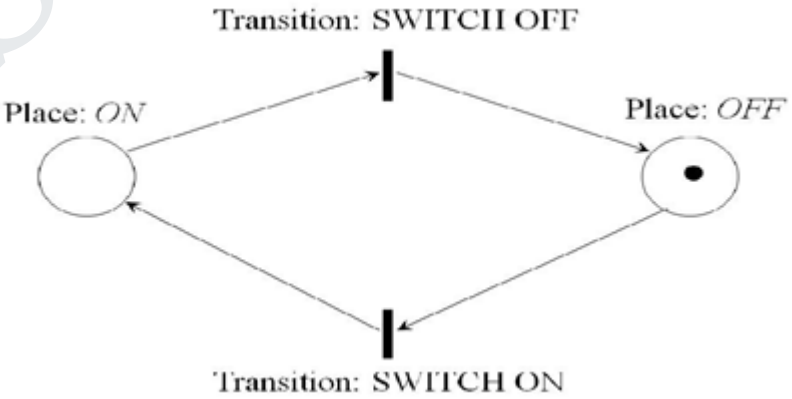
Here it is, written in Z

$$\forall p : PERSON \bullet$$
$$(\exists s1, s2 : SHOE \bullet wears(p, s1, s2))$$
$$\wedge\ (\forall d : DOG \bullet isWith(p, d) \Rightarrow carries(p, d))$$

A Petri Nets (PN) comprises places, transitions, and arcs

- Places are system states
- Transitions describe events that may modify the system state
- Arcs specify the relationship between places

Tokens reside in places, and are used to specify the state of a PN



- Two places: Off and On
- Two transitions: Switch Off and Switch On
- Four arcs

- The off condition is true
- A transition can fire if an input token exists
  - One token is moved from the input place to the output

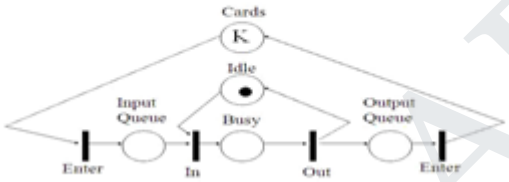
place.PN properties

- 8-tuple mathematical model

– M={P,T,I,O,H,PAR,PRED,MP}
– P - the set of places
– T - the set of transitions
– I,O,H - Input, output, inhibition function
– PAR - the set of parameters
– PRED - Predicates restricting parameter range
– PM - Parameter value

- From this linear algebra can be used to analyze a network

Manufacturing Example



- Very rich modeling
- Easily capable of modeling software project, requirements, architectures, and processes
- Drawbacks
  - Complex rules
  - Analysis quite complex

Data Dictionary

Provides definitions for all elements in the system which include:

- Meaning of data flows and stores in DFDs
- Composition of the data flows e.g. customer address breaks down to street number, street name, city and postcode
- Composition of the data in stores e.g. in Customer store include name, date of birth, address, credit rating etc.

Details of the relationships between entities

Data dictionary Notation

=is composed of		
	+	and
( )		optional (may be present or absent)
{ }		iteration
[ ]		select one of several alternatives
	**	comment
	@	identifier (key field) for store
|		separates alternative choices in the [ ] construct

Data dictionary Examples

name = courtesy-title + first-name + (middle-name) + last-name courtesy-title = [Mr. | Miss | Mrs. | Ms. | Dr. | Professor] first-name = {legal-character}

middle-name ={legal-character} last-name = {legal-character} legal-character = [A-Z|a-z|0-9|‘|-| |]
Current-height =** *units: metres; range: 1.00-2.50*
sex =***values: [M|F]*

As both are elementary data, no composition need be shown, though an explanation of the relevant units/symbols is needed order = customer-name + shipping-address + 1{item}10 means that an order always has a customer name and a shipping address and has between 1 and 10 items.

UNIT III
SOFTWARE DESIGN

Design process – Design Concepts-Design Model– Design Heuristic – Architectural Design – Architectural styles, Architectural Design, Architectural Mapping using Data Flow- User Interface Design: Interface analysis, Interface Design –Component level Design: Designing Class based components, traditional Components

3.1. DESIGN PROCESS

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

Design is represented at a high level of abstraction - a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

3.1.1. SOFTWARE QUALITY GUIDELINES AND ATTRIBUTES

Three characteristics that serve as a guideline for the evaluation of a good design.

- The design must implement all of the explicit requirements.
- The design must be a readable, understandable.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains.

3.1.2. QUALITY GUIDELINES

Design concepts also serve as software quality criteria. Consider the following guidelines.

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

3.1.3. QUALITY ATTRIBUTES

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

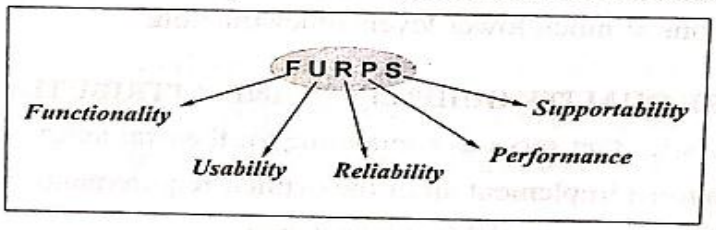


Fig. 3.1. FURPS Quality Attributes

- **Functionality:** It is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- **Usability** It is assessed by considering overall aesthetics, consistency, and documentation.
- **Reliability:** It is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- **Performance:** It is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability:** It combines the ability to extend the program (extensibility), adaptability, serviceability.

3.2. DESIGN CONCEPTS

Software design is a process of problem solving and planning for a software solution.

Requirements are translated into a blueprint for constructing the software.

1. Conceptual design or system design tells the customer what the system will do.
2. Technical design allows the system builders to understand the hardware and software needed to solve the customers.

Design Concepts

1. Abstraction
2. Architecture
3. Patterns
4. Modularity
5. Information hiding
6. Functional Independence
7. Refinement
8. Refactoring
9. Design Classes

3.2.1. ABSTRACTION

It concentrates on the essential features and ignores details that are not relevant.

1. Procedural abstraction – a named sequence of instruction that has a specific and limited function.
2. Data abstraction – a named collection of data that describes a data object.
3. Control abstraction – implies a program control mechanism without specifying internal details.

3.2.2. ARCHITECTURE

Architecture is a defined as a structure and organization of program components (modules).

5 different types of models are used to represent the architectural design.

(*i*) **Structural models:** represent architecture as an organized collection of program components.

(*ii*) **Framework models:** identify repeatable architectural design framework that similar to the types of applications

(*iii*) **Dynamic models:** behavioral aspects of the program architecture

(*iv*) **Process models:** design of the business or technical process of a system

(*v*) **Functional models:** functional hierarchy of a system

3.2.3. PATTERNS

- A design structure that solves a particular length problem within a specific context.
- It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns.

3.2.4. MODULARITY

System is decomposed into a number of modules.

Modularity = Abstraction + Partitioning

By "***Divide and conquer***" strategy, solve complex problem by breaking into pieces.

Consider 2 Problem P1 and P2. C(X) defines complexity of the problem and E(X) defines the effort or time required to solve the problem X.

$$C(P1 + P2) > C(P1) + C(P2)$$

$$E(P1 + P2) > E(P1) + E(P2)$$

Therefore sum of complexity of 2 problems is greater than individual complexity of problem.

There are five criteria to evaluate a design method with respect to its ability to define effective modular system.

1. **Modular Decomposability**

   Provides a systematic approach for decomposing the problem into subproblems.

2. **Modular Composability**

   Enables existing (reusable) design components to be assembled into a new system.

3. **Modular Understability**

Module can be understood as a stand-alone unit (no need to refer to other modules).

4. **Modular Continuity**

Small changes to the system requirements result in changes to individual modules.

5. **Modular protection**

Unexpected condition occurs within a module and its effects are constrained within that module.

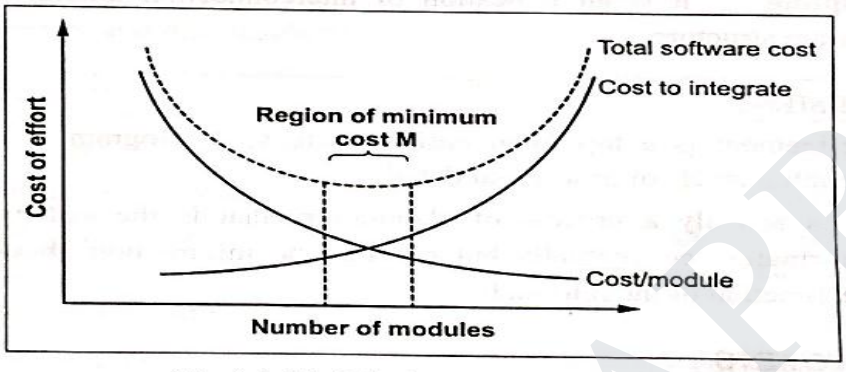


Fig. 3.2. Modularity and Software cost

3.2.5. INFORMATION HIDING

Information (Procedure and Data) contained within a module is inaccessible to other modules.

Elements of Information Hiding

- Data abstraction
- Format of contral blocks
- Character Codes and implementations details
- Shifting, masking and small other machine dependent details.

3.2.6. FUNCTIONAL INDEPENDENCE

- Develop module which address specific sub function of requirement.
- Independent module
  - Easier to maintain and test
  - Error propagation reduced
  - Reusable modules are possible
- Modules that have a "single-minded" function and an aversion to excessive interaction with other modules.
- **High cohesion** – a module performs only a single task. Cohesion is an indication of the relative functional strength of a module.
- **Coupling** – It is an indication of interconnection among modules in a software structure.

3.2.7. REFINEMENT

Stepwise refinement is a top-down design strategy. A program is developed by successively refining levels of procedural detail.

Refinement is actually a process of elaboration, that is the statement describes function or information conceptually but provides no information about the internal workings of the function of the information.

3.2.8. REFACTORING

A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behavior

- Removes redundancy
- unused design elements
- inefficient or unnecessary algorithms
- poorly constructed or inappropriate data structures
- or any other design failures

3.2.9. DESIGN CLASSES

- Describes element of the problem domain
- Creates a new set of design classes that implement a software infrastructure to support the business solution
- **User interface classes** – define all abstractions necessary for human-computer interaction.
- **Business domain classes** – identify attributes and services (methods) that are required to implement some element of the business domain
- **Process classes** – implement business abstractions required to fully manage the business domain classes
- **Persistent classes** – represent data stores (***Example:*** a database).
- **System classes** – implement software management and control functions that enable the system to operate and communicate within its computing environment and the outside world

3.3. DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in figure. The process dimension indicates the evolution of the parts of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

In the figure 3.3, the dashed line indicates the boundary between the analysis and design models. In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious.

The elements of the design model use many of the same UML diagrams that were used in the analysis model.

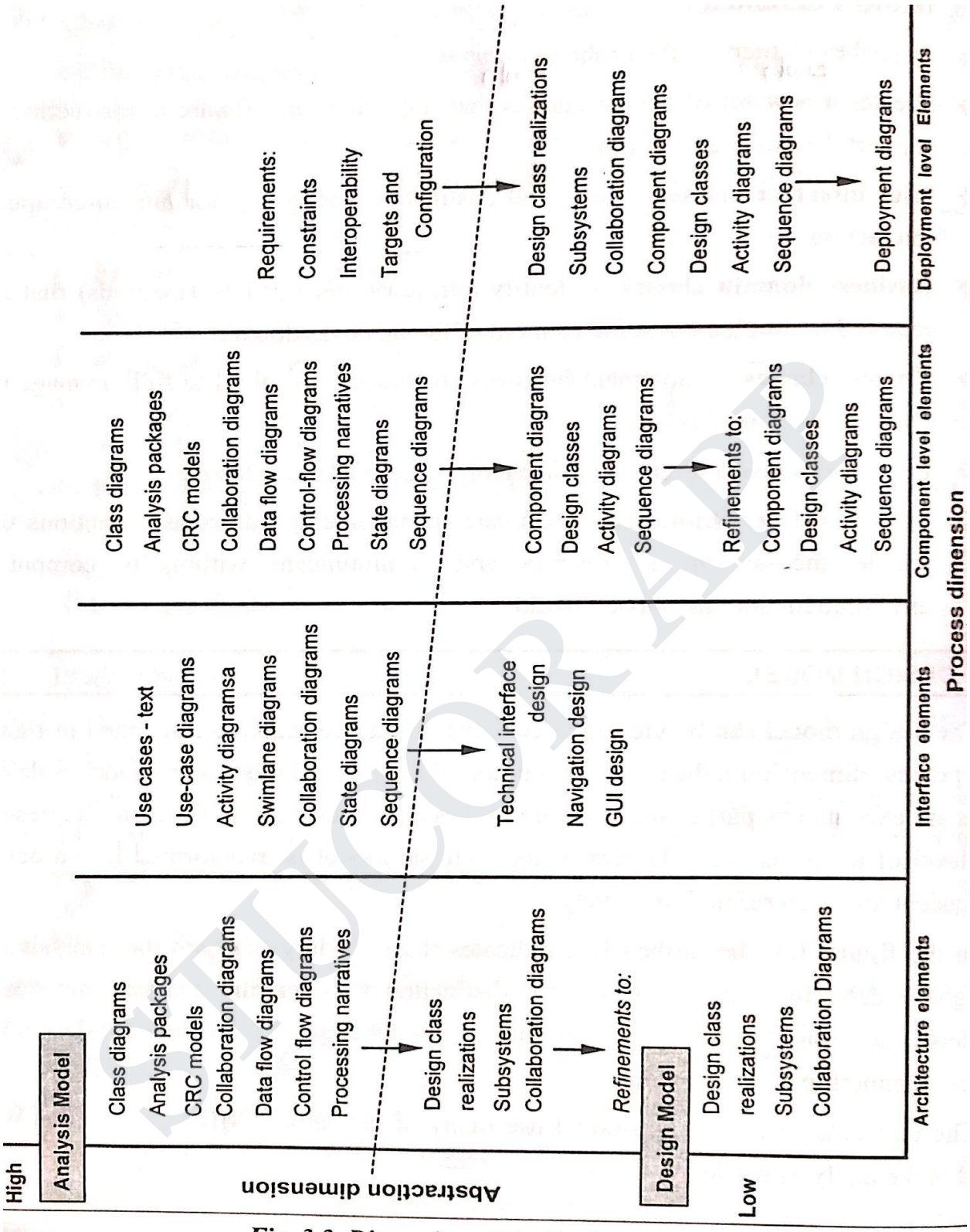


Fig. 3.3. Dimension of the Design Model

3.3.1. DATA DESIGN ELEMENTS

Data Design sometimes referred to as ***data architecting***, creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

3.3.2. ARCHITECTURAL DESIGN ELEMENTS

The architectural design for software is equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

The architectural model is derived from three sources.

1. Information about the application domain for the software to be built
2. Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand and
3. The availability of architectural styles and patterns

3.3.3. INTERFACE DESIGN ELEMENTS

The interface design for software is analogous to a set of detailed drawings for the doors, windows, and external utilities of a house.

There are three important elements of interface design:

1. The User Interface (UI)
2. External interfaces to other systems, devices, networks, or other producers or consumers of information and
3. Internal interfaces between various design components.

These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

For example, the Safe Home security function makes use of a control panel that allows a homeowner to control certain aspects of the security function.

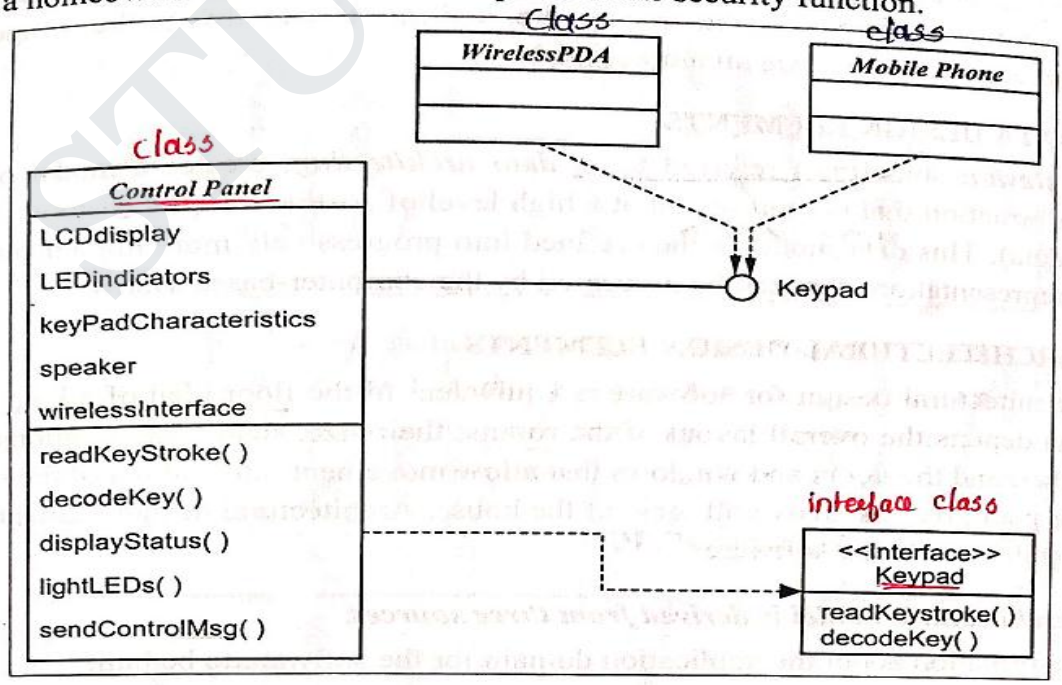


Fig. 3.4. Interface representation for Control-Panel

The dashed line with an open triangle at its end indicates that the Control Panel class provides KeyPad operations as part of its behavior.

3.3.4. COMPONENT-LEVEL DESIGN ELEMENTS

The component-level design for software is the equivalent to a set of detailed drawings and each specification for each room in a house.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations.

A component is represented in UML diagrammatic form as shown in Figure 3.5.

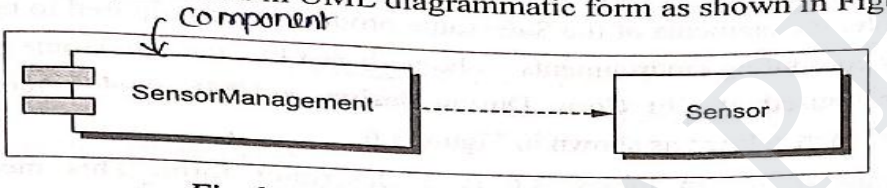


Fig. 3.5. A UML Component Diagram

In this figure, a component named SensorManagement is represented. A dashed arrow connects the component to a class named Sensor that is assigned to it. The SensorManagement component performs all functions associated with SafeHome sensors including monitoring and configuring them.

3.3.5. DEPLOYMENT-LEVEL DESIGN ELEMENTS

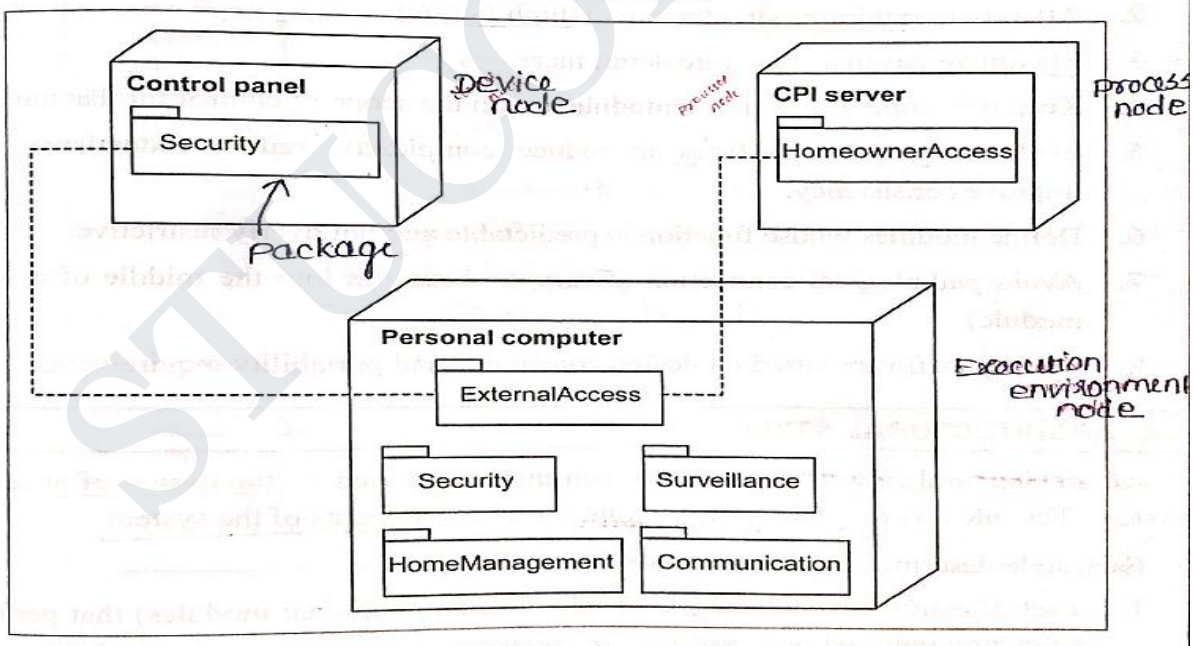


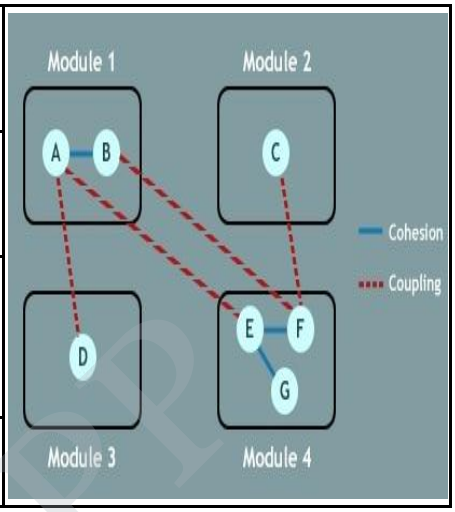
Fig. 3.6. A UML Deployment Diagram

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

For example, the elements of the SafeHome product are configured to operate within three primary computing environments - a home-based PC, the SafeHome control panel, and a server housed at CPI Corp. During design, a UML deployment diagram is developed and then refined as shown in Figure 3.6.

3.4 DESIGN HEURISTICS

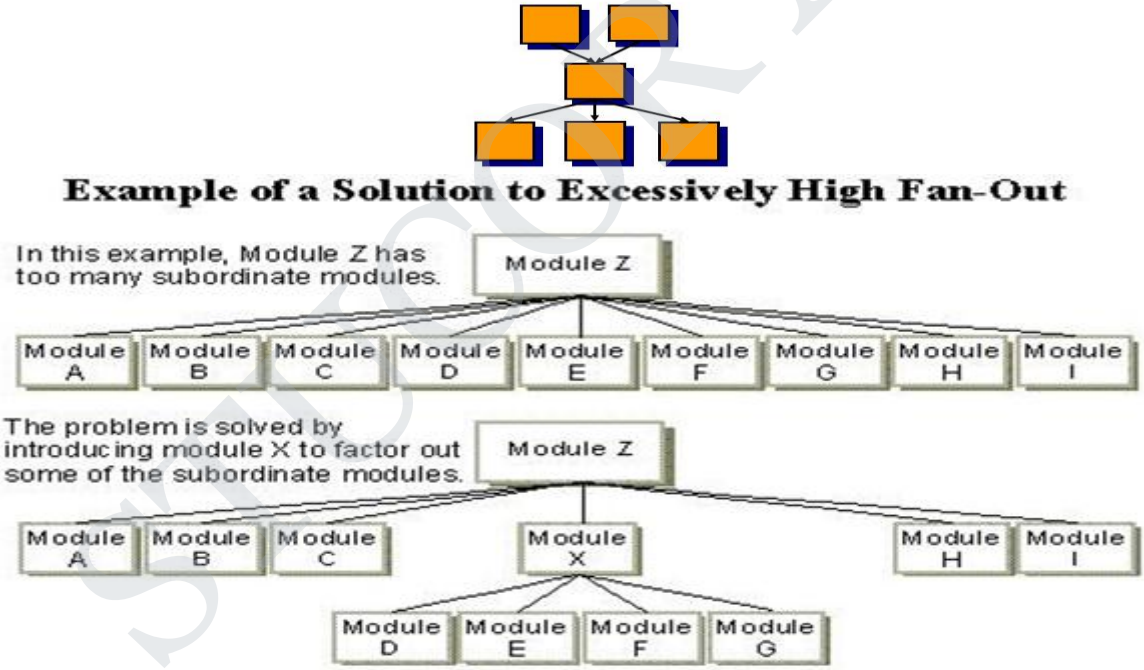
1. Evaluate the first iteration of the program structure to **reduce coupling and improve cohesion**.

Cohesion is the indication of the relationship within **module**.	**Coupling** is the indication of the relationships between modules.	
Cohesion shows the module's relative **functional dependence** strength.	Coupling shows the relative **independence** among the modules.	
While designing you should strive for **high cohesion** .i.e. a cohesive component/ module focus on a single task	While designing you should strive for **lowcoupling** i.e. **dependency** between modules should be less.	
Cohesion is **Intra module** Concept.	Coupling is **Inter - Module** Concept.	

2)Attempt to **minimize** structures with high **fan-out**

3)Maximize **fan-in** as structure depth **increases.**

Fan-in = number of ingoing dependencies
Fan-out = number of outgoing dependencies



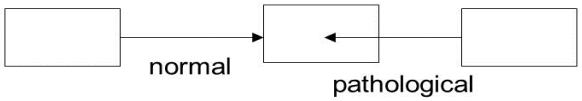
4)Keep **the scope of effect of a module** within the scope of control for that module.

5)Evaluate **module interfaces** to reduce complexity, reduce redundancy, and improve consistency.

6)**Define modules** whose function is predictable.

7)Strive for **"controlled entry"** modules by avoiding **"pathological connections."** (e.g. branches into the middle of another module)

- Connections



8)Package software based on **design constraints and portability requirements**.

3.5. ARCHITECTURAL STYLE

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

Each style describes a system category that encompasses

1. a set of components (***Example***, databases, computational modules) that perform a function required by a system
2. a set of connectors that enable "communication, coordination and cooperation" among components
3. constraints that define how components can be integrated to form the system; and
4. semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts

The numbers of architectural styles are:

3.5.1. DATA CENTERED ARCHITECTURE

A data store (Example 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.

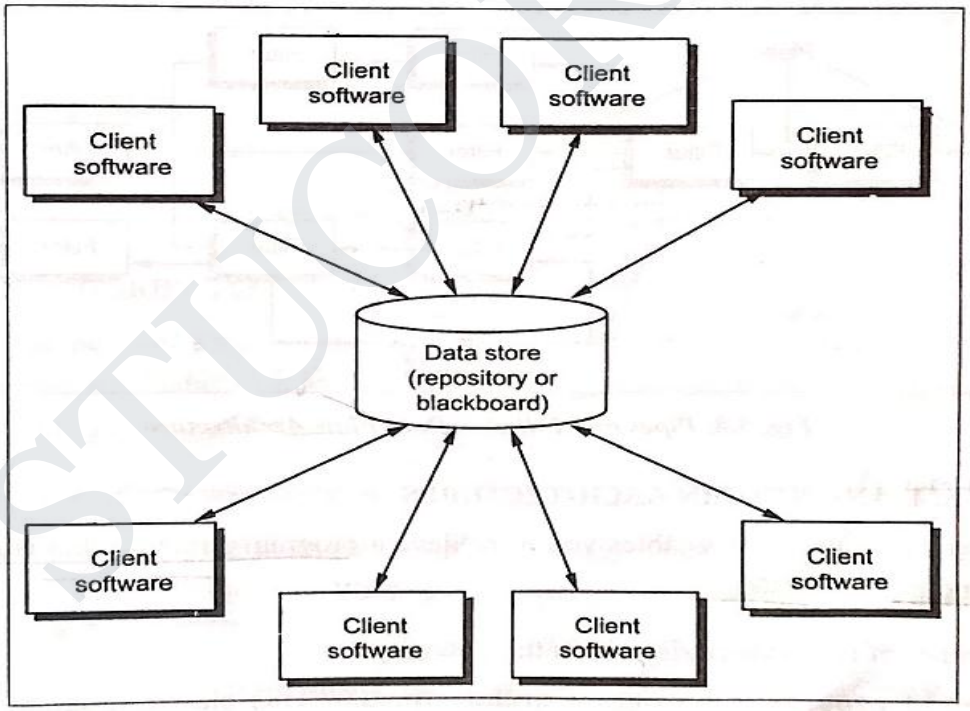


Fig. 3.7. Data Centered Architecture

Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients as clients operate independently. In addition, data can be passed among clients using the blackboard mechanism which serves to coordinate the transfer of information between clients.

3.5.2. 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.

If the data flow degenerates into a single line of transforms, it is termed batch sequential.

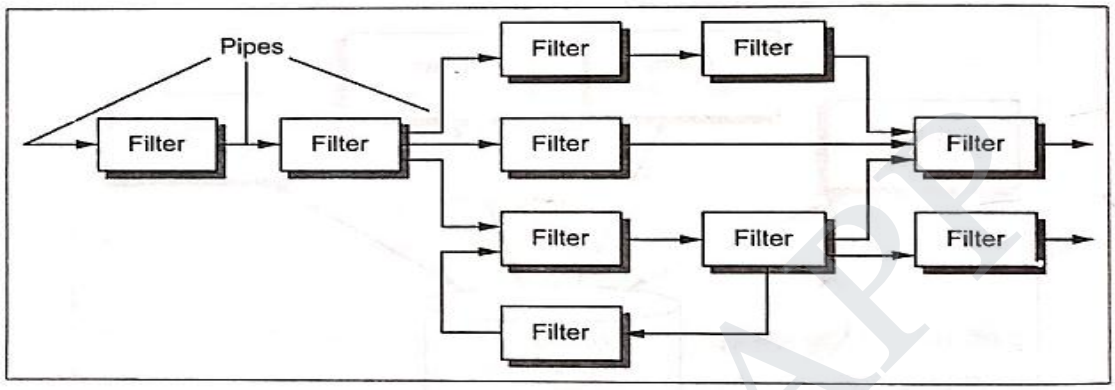


Fig. 3.8. Pipes and Filters – Data Flow Architecture

3.5.3. CALL AND RETURN ARCHITECTURES

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

A number of sub-styles exist within this category:

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

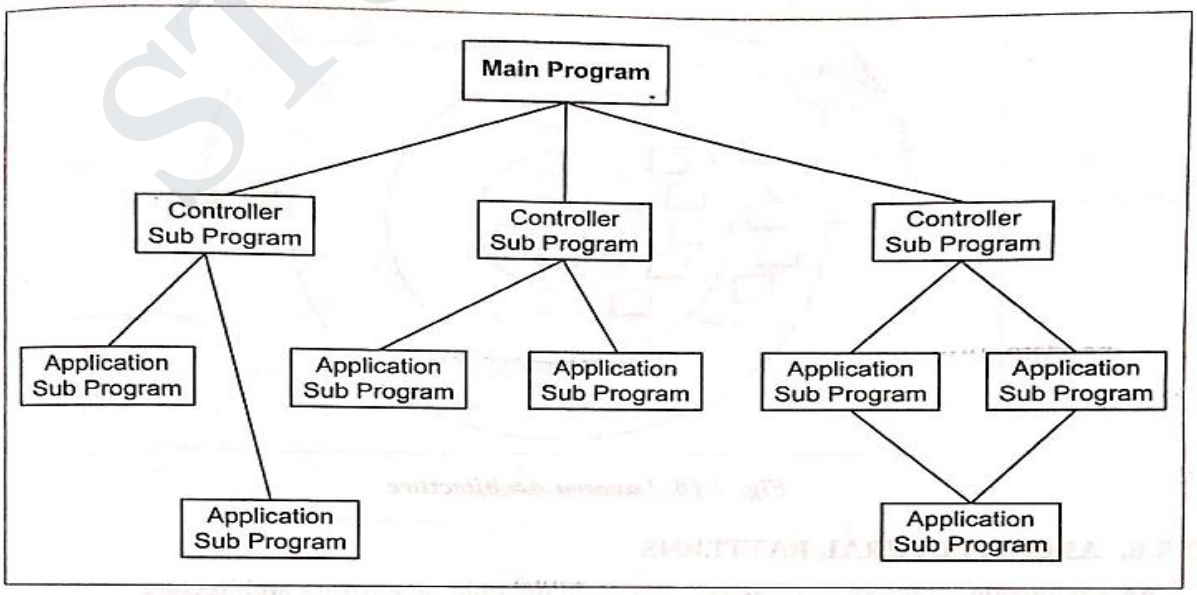


Fig. 3.9. Main Program/Sub program Architecture

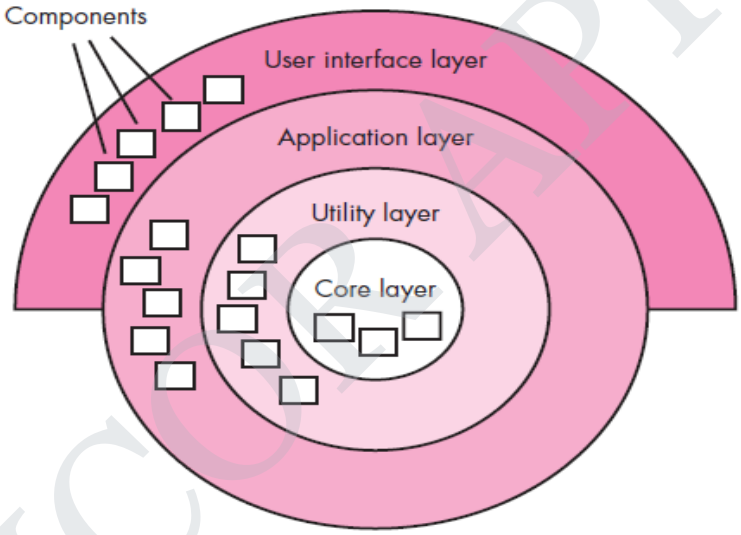
3.5.4. OBJECT-ORIENTED ARCHITECTURES

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

3.5.5. LAYERED ARCHITECTURE

The basic structure of a layered architecture has 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.



Layered architecture

3.5.6. ARCHITECTURAL PATTERNS

An Architectural pattern is a standard design in the field of software architecture. Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

3.6 ARCHITECTURE DESIGN

The architectural design begins the software to be developed must be put into context which defines the external entities that the software interacts with and the nature of the interaction.

3.6.1. REPRESENTING THE SYSTEM IN CONTEXT

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

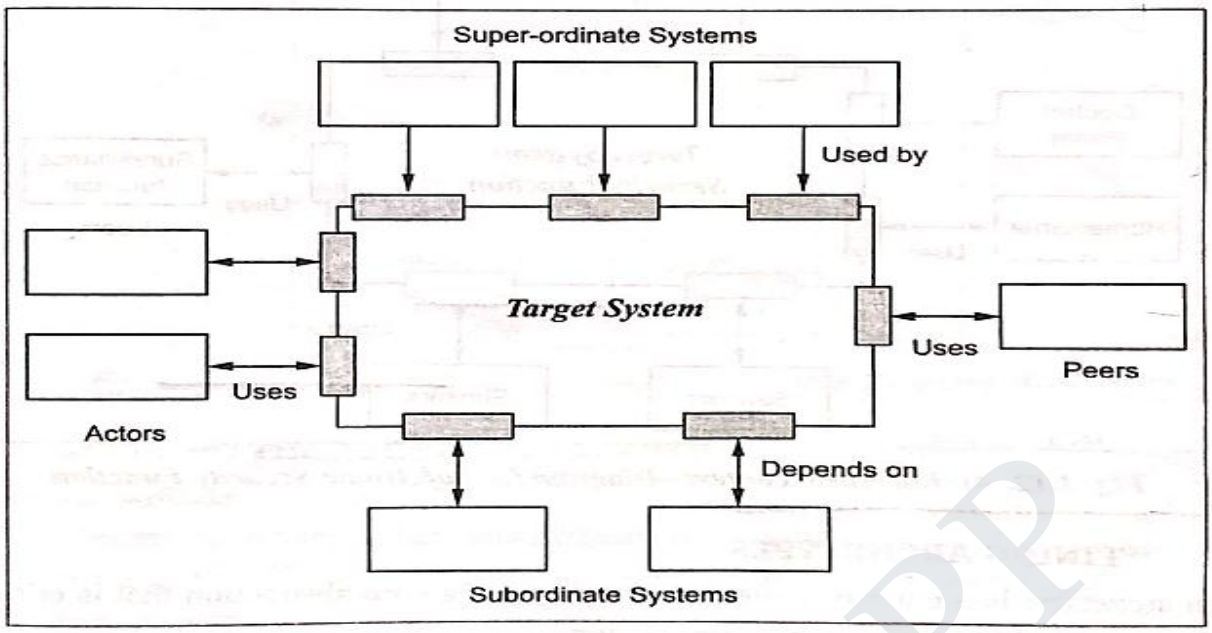


Fig. 3.11. Architectural Context Diagram

The generic structure of the architectural context diagram is illustrated in Figure 3.11 and systems that interoperate with the target system are represented as

- **Superordinate systems** – those systems that use the target system as part of some higher-level processing scheme.
- **Subordinate systems** – those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- **Peer-level systems** – those systems that interact on a peer-to-peer basis, information is either produced or consumed by the peers and the target system.
- **Actors** – entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

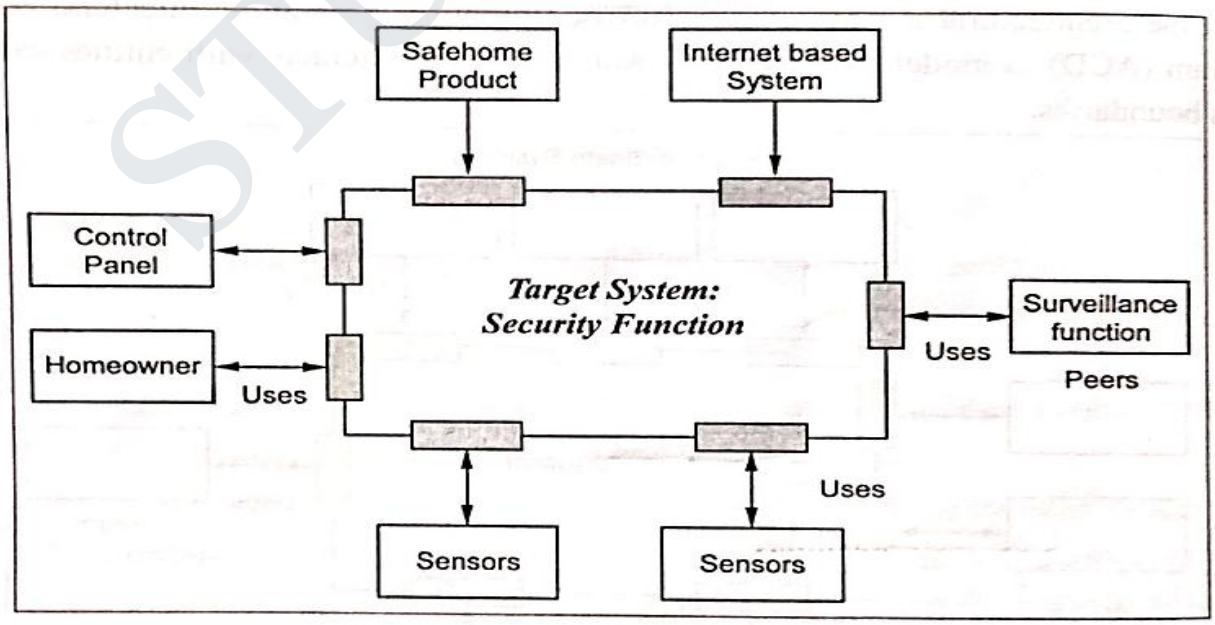


Fig. 3.12. Architectural context –Diagram for SafeHome Security Function

3.6.2. DEFINING ARCHETYPES

An archetype is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target system.

The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

The Safehome Security function defines the following archetypes.

- **Node** – Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.
- **Detector** – An abstraction that encompasses all sensing equipment that feeds information into the target system.
- **Indicator** – An abstraction that represents all mechanisms (Example, alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **Controller** – An abstraction that depicts the mechanism that allows the arming or disarming of a node.

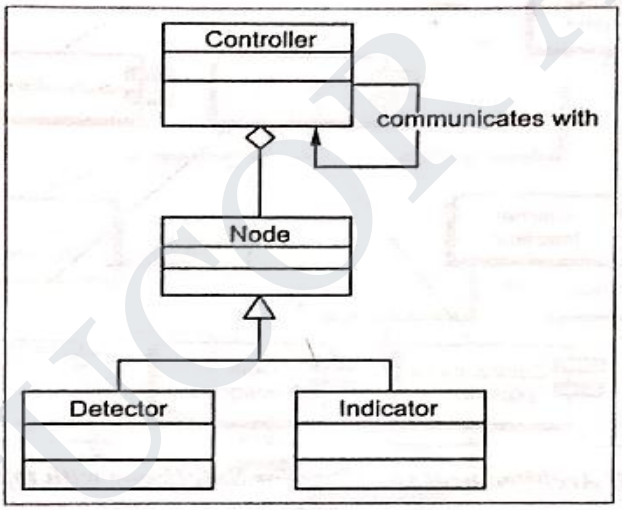


Fig. 3.13. UML Relationships for SafeHome Security Function Archetypes

3.6.3. REFINING THE ARCHITECTURE INTO COMPONENTS

As the software architecture is refined into components, the the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain.

Example of SafeHome Security Function – The top level components are,

- **External communication management** – coordinates communication of the security function.
- **Control panel processing** – manages all control panel functionality.
- **Detector management** – coordinates access to all detectors attached to the system.
- **Alarm processing** – verifies and acts on all alarm conditions.

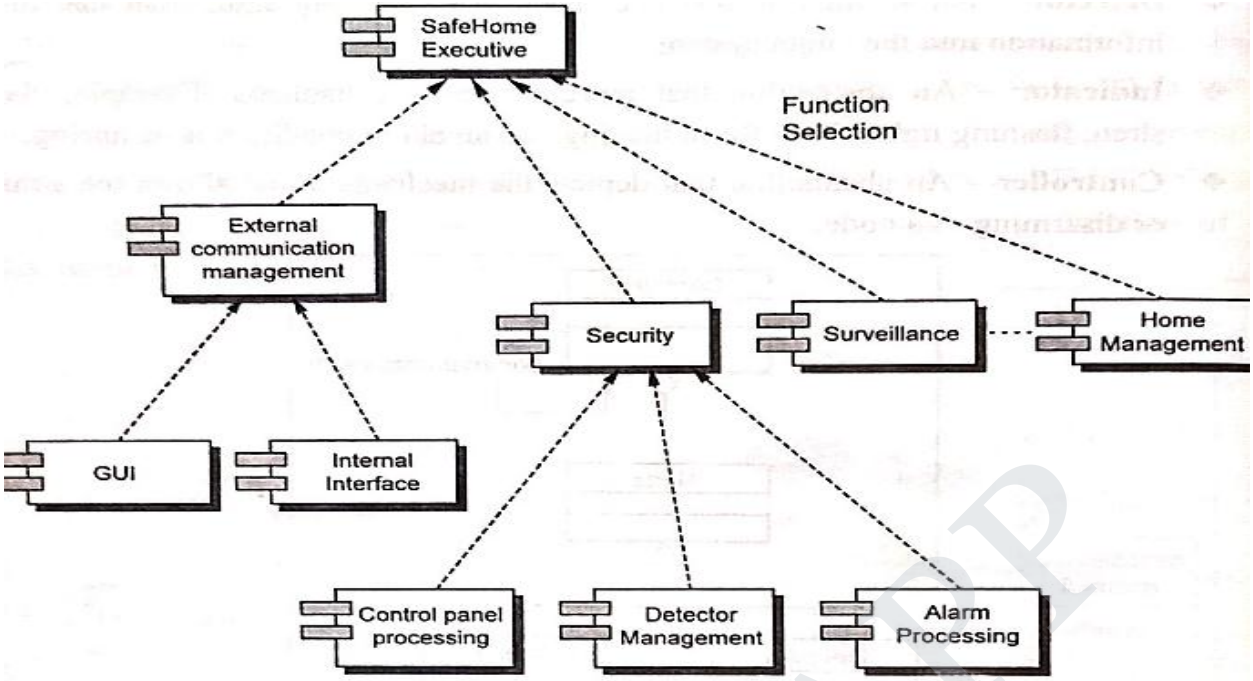


Fig. 3.14. Overall Architectural structure for SafeHome with top-level components

3.6.4. DESCRIBING INSTANTIATION OF THE SYSTEM

An actual instantiation of the architecture is developed. The architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

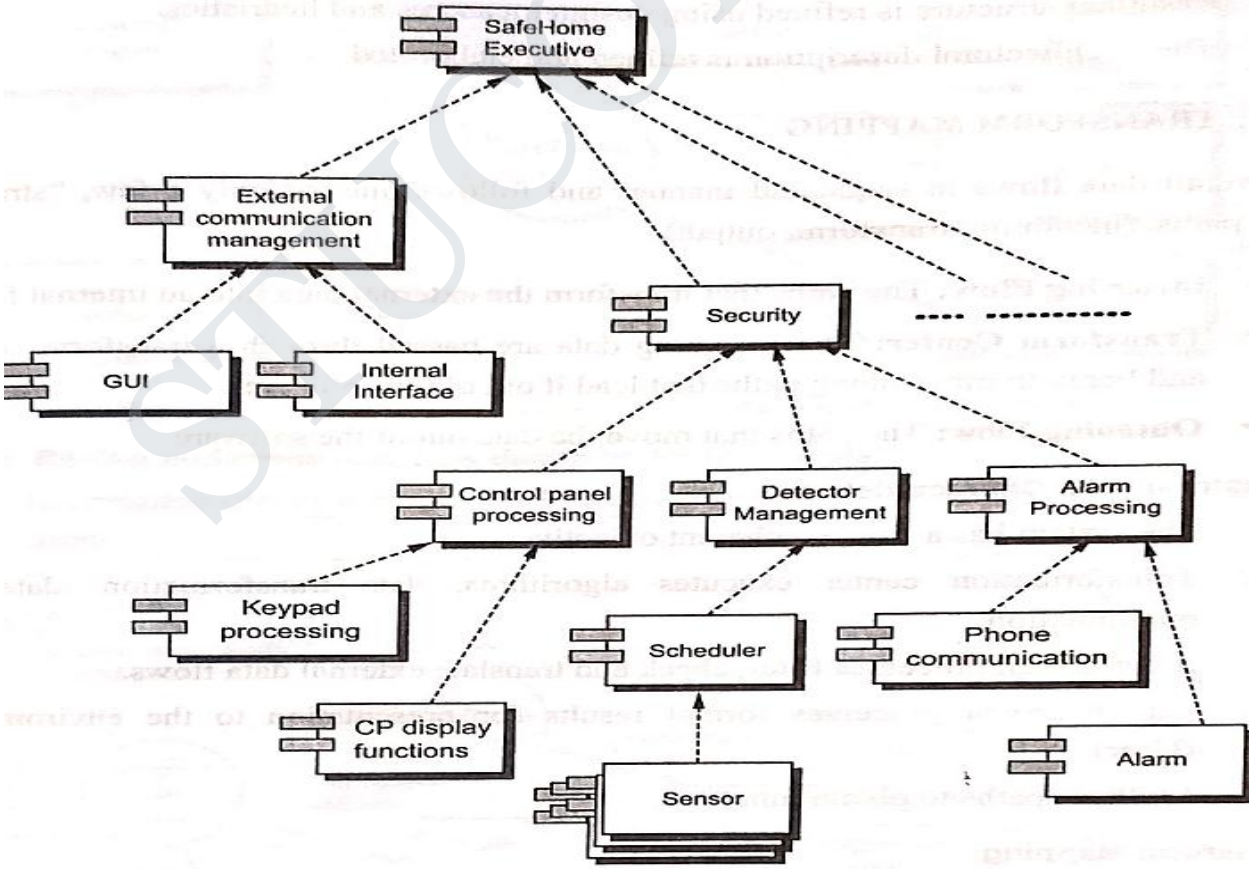


Fig. 3.15. An instantiation of the security function with component elaboration

3.7 MAPPING DATA FLOW INTO A SOFTWARE ARCHITECTURE

- Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.

Example: SafeHome Security System.

3.7.1.1. DESIGN STEPS

Step 1. Review the fundamental system model

- Depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function.

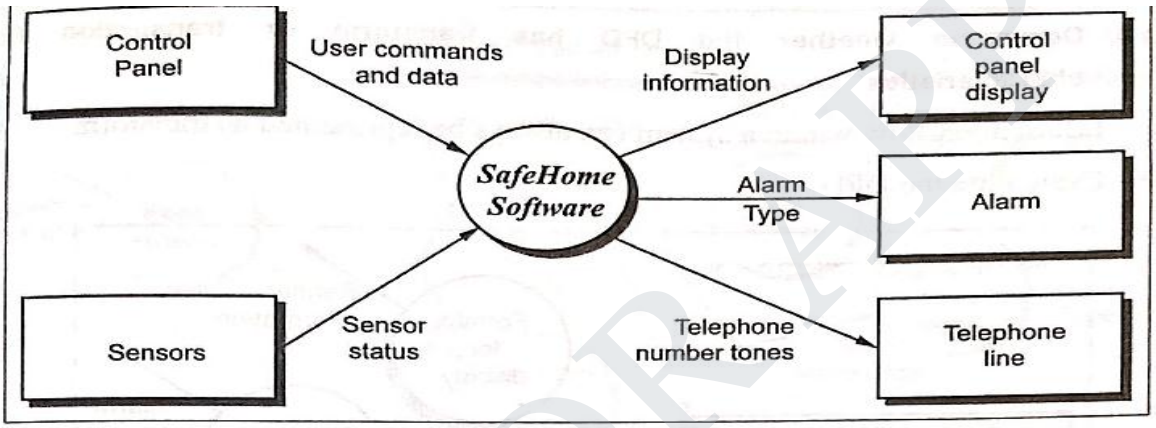


Fig. 3.16. Review the fundamental system for SafeHome Software

Step 2: Review and refine data flow diagrams for the software.

- Information obtained from the requirements model is refined to produce greater detail.

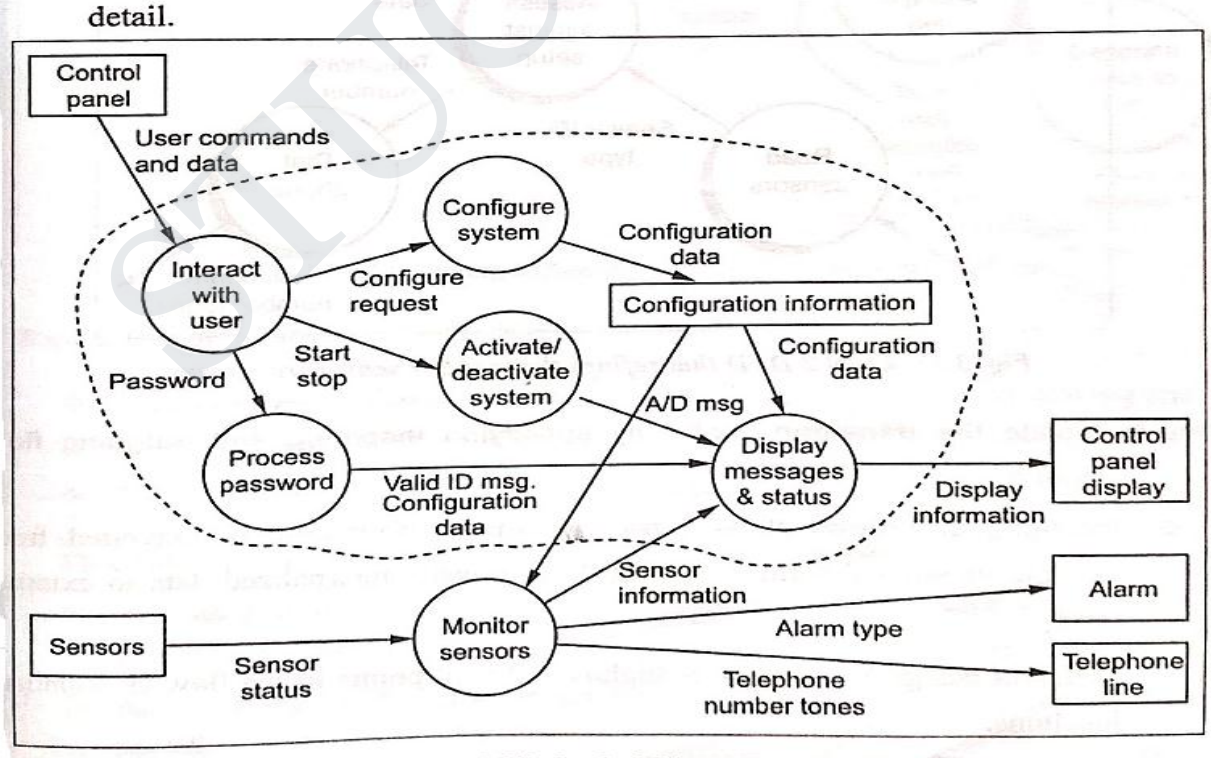


Fig. 3.17. Level 1 DFD for SafeHome Security Function

Step 3: Determine whether the DFD has transform or transaction flow characteristics

- Information flow within a system can always be represented as transform
- Evaluating the DFD

Step 4: Isolate the transform center by specifying incoming and outgoing flow boundaries.

- Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form.
- Different designers may select slightly different points in the flow as boundary locations.

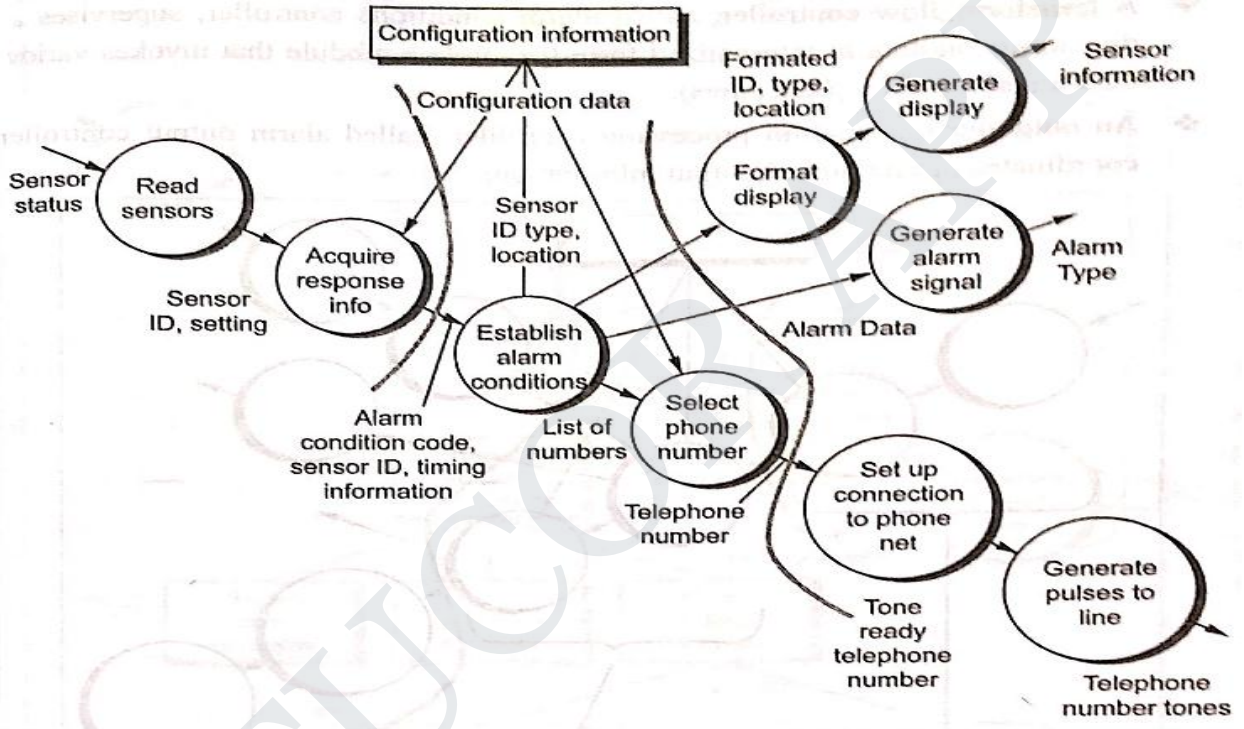


Fig. 3.19. Specifying incoming and outgoing flow boundaries

Step 5: Perform "first-level factoring"

- Program structure in which top-level modules perform decision making and low level modules perform most input, computation, and output work.
- Number of modules is limited to minimum.

This first-level factoring is for the monitor sensors subsystem.

- A main controller (called monitor sensors executive) resides at the top of the program structure and coordinates the following subordinate control functions:
- An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.
- A transform flow controller, called alarm conditions controller, supervises all operations on data in internalized form (example a module that invokes various data transformation procedures).
- An outgoing information processing controller, called alarm output controller, coordinates production of output information.

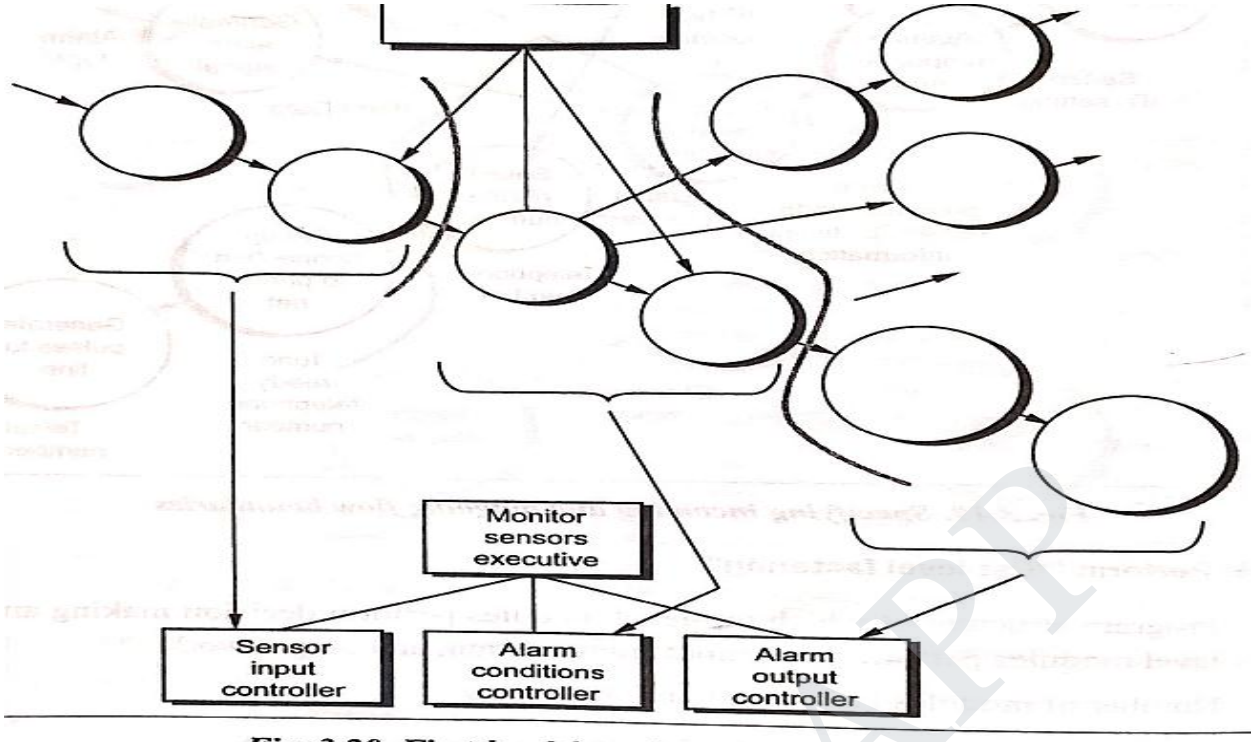


Fig. 3.20. First-level factoring for monitor sensors

Step 6. Perform "second-level factoring"

- Mapping individual transforms (bubbles) into appropriate modules.
- Factoring is accomplished by moving outward from the transform center boundary.

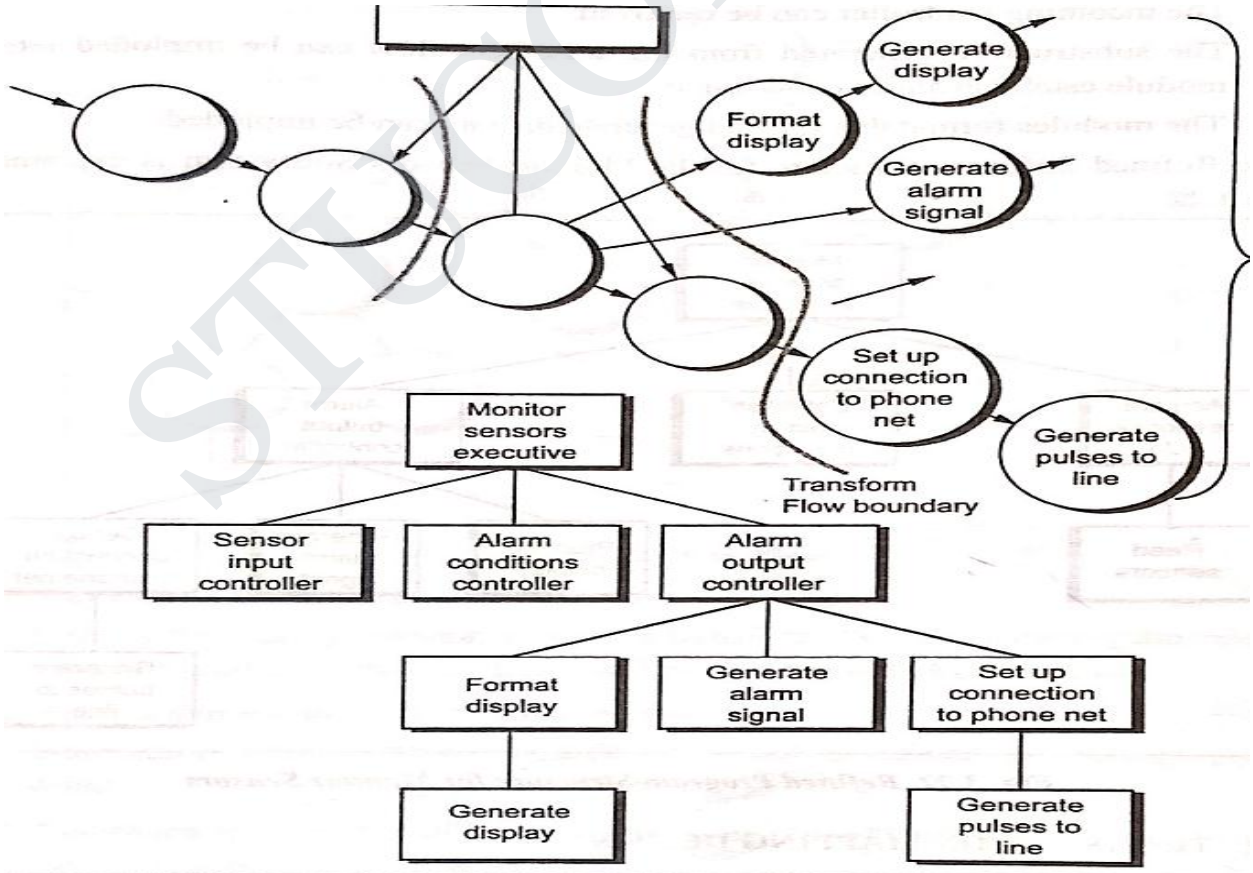


Fig. 3.21. Second-level factoring for monitor sensors

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

- First-iteration architecture can always be refined by applying concepts of module independence.
- Many modifications can be made to the first iteration architecture developed for the SafeHome monitor sensors subsystem.

1. The incoming controller can be removed.
2. The substructure generated from the transform flow can be imploded into the module establish alarm conditions.
3. The modules format display and generate display can be imploded.

The Refined Software Structure for the Monitor Sensors Subsystem is explained in Figure 3.22.

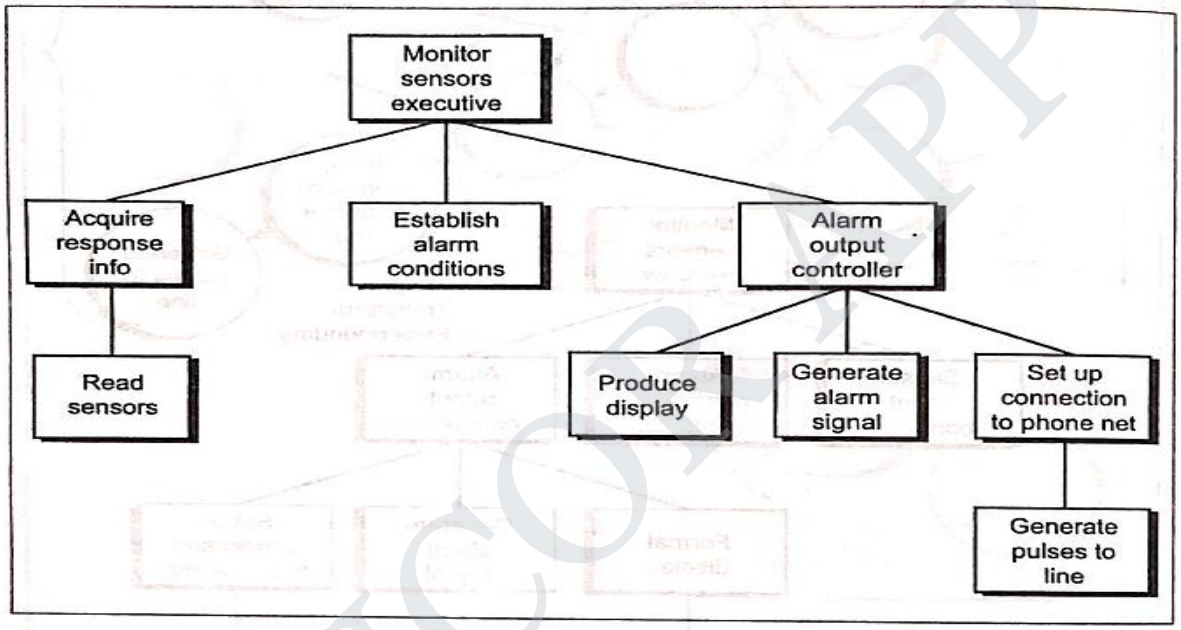


Fig. 3.22. Refined Program Structure for Monitor Sensors

3.8. USER-INTERFACE DESIGN

User Interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for an user interface prototype.

Interface design focuses on three areas of concern:

1. The design of interfaces between software components.
2. The design of interfaces between the software and other nonhuman producers and consumers of information
3. The design of the interface between a human and the computer.

3.8.1. THE GOLDEN RULES

Theo Mandel coins three "golden rules".

1. Place the User in Control
2. Reduce the User's Memory Load
3. Make the Interface Consistent

3.8.1.1. Place the User in Control

Mandel defines a number of design principles that allow the user to maintain control.

1. **Define interaction modes in a way that does not force a user into unnecessary or undesired actions**

   An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode.

2. **Provide for flexible interaction**

   For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen and voice recognition commands.

3. **Allow user interaction to be interruptible and undoable**

   The user should also be able to "undo" any action.

4. **Streamline interaction as skill levels advance and allow the interaction to be customized**

   Users often find that they perform the same sequence of interactions repeatedly. Design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

5. **Hide technical internals from the casual user**

   The user should not be aware of the operating system, file management functions, or other computing technology.

6. **Design for direct interaction with objects that appear on the screen**

   For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

3.8.1.2. Reduce the User's Memory Load

Mandel defines design principles that enable an interface to reduce the user's memory load

1. **Reduce demand on short-term memory**

   The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

2. **Establish meaningful defaults**

   The initial set of defaults should make sense for the average user.

3. **Define shortcuts that are intuitive**

   When mnemonics are used to accomplish a system function (example, alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember.

4. **The visual layout of the interface should be based on a real-world metaphor**

   For example, a bill payment system should use a check book and check register metaphor to guide the user through the bill paying process.

5. **Disclose information in a progressive fashion**

   The interface should be organized hierarchically.

3.8.1.3. Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that

1. All visual information is organized according to design standard that is maintained throughout all screen displays,

2. Input mechanisms are constrained to a limited set that is used consistently throughout the application, and
3. mechanisms for navigating from task to task are consistently defined and implemented.

1. **Allow the user to put the current task into a meaningful context.**
   It is important to provide indicators (Example, window titles, graphical icons consistent colour coding) that enable the user to know the context of the work at hand.
2. **Maintain consistency across a family of application**
   A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.
3. **If past interactive models have created user expectations,** do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (Example: the use of alt-S to save a file), the user expects this in every application he encounters. A change (Example: using alt-S to invoke scaling) will cause confusion.

3.9. USER INTERFACE ANALYSIS AND DESIGN

The overall process for analyzing and designing a user interface begins with the creation of different models of system function.

3.9.1. INTERFACE ANALYSIS AND DESIGN MODELS

Four different models come into play when a user interface is to be analyzed and designed. A human engineer establishes a user model, the software engineer designs a design model, the end user develops a mental image that is called the user's mental model or the system perception, and the implementers of the system create an implementation model.

Users can be categorized as:

- **Novices** – No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.
- **Knowledgeable, intermittent users** – Reasonable semantic knowledge of the application.
- **Knowledgeable, frequent users** – Good semantic and syntactic knowledge that often leads to the "power-user syndrome".

3.9.2. THE PROCESS

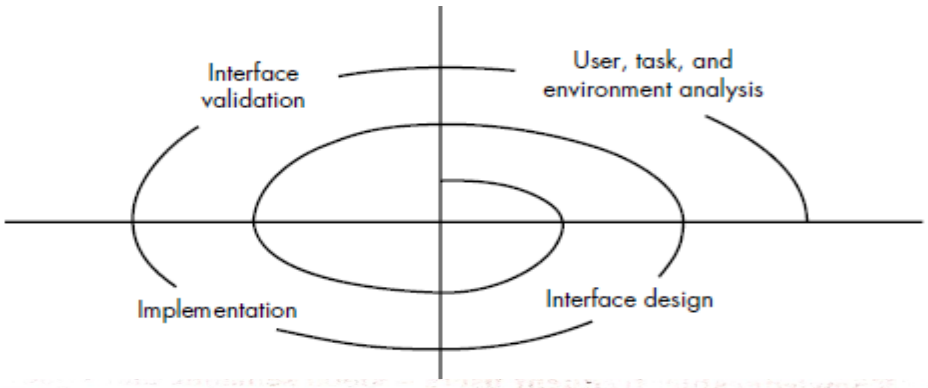
The user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities:

1. Interface analysis and modeling
2. Interface design
3. Interface construction, and
4. Interface validation.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded and different user categories are defined.

The goal of interface design is to define a set of interface objects and actions that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

The user interface design process



Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated.

Interface validation focuses on

1. The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;
2. The degree to which the interface is easy to use and easy to learn, and
3. The users' acceptance of the interface as a useful tool in their work.

3.10. INTERFACE ANALYSIS

A key tenet of all software engineering process models is this: understand the problem before you attempt to design a solution. Understanding the problem 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 and
4. The environment in which these tasks will be conducted

3.10.1. USER ANALYSIS

User interface is probably all the justification needed to spend some time understanding the user before worrying about technical matters.

Information from a broad array of sources are,

- **User Interviews** – The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues.
- **Sales input** – Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.
- **Marketing input** – Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.
- **Support input** – Support staff talks with users on a daily basis.

3.10.2. TASK ANALYSIS AND MODELING

The goal of task analysis is to answer 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?

- **Uses Cases** – It describes the manner in which an actor interacts with a system.
- **Task Elaboration** – Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.

- **Object Elaboration** – Examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations.
- **Workflow analysis** – When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis
- **Hierarchical representation** – Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user.

For example, consider the following user task and subtask hierarchy.

User task: Requests that a prescription be refilled

- Provide identifying information
  - ➢ Specify name
  - ➢ Specify userid
  - ➢ Specify PIN and password
- Specify prescription number
- Specify date refill is required

3.10.4. ANALYSIS OF THE WORK ENVIRONMENT

In some applications the user interface for a computer-based system is placed in a "user-friendly location" (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

3.11 INTERFACE DESIGN ACTIVITIES

Once task analysis has been completed, all tasks (or objects and actions) required by the end-user have been identified in detail and the interface design activity commences. The first interface design steps can be accomplished using the following approach:

1. Establish the goals and intentions for each task.

2. Map each goal and intention to a sequence of specific actions.

3. Specify the action sequence of tasks and subtasks, also called a *user scenario*, as it will be executed at the interface level.

4. Indicate the state of the system; that is, what does the interface look like at the time that a user scenario is performed?

5. Define control mechanisms; that is, the objects and actions available to the user to alter the system state.

6. Show how control mechanisms affect the state of the system.

7. Indicate how the user interprets the state of the system from information provided through the interface.

3.11.1 DEFINING INTERFACE OBJECTS AND ACTIONS

An important step in interface design is the definition of interface objects and the actions that are applied to them. To accomplish this, the user scenario is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Example – User Scenario of SafeHome System. The use case is represented as follows.

- accesses the SafeHome system
- enters an ID and password to allow remote access
- checks system status
- arms or disarms SafeHome system
- displays floor plan and sensor locations
- displays zones on floor plan
- changes zones on floor plan
- displays video camera locations on floor plan
- selects video camera for viewing
- views video images
- pans or zooms the video camera

3.11.3. DESIGN ISSUES

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling.

Response time – System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action until the software responds with desired output or action.

Help facilities – Almost every user of an interactive, computer-based system requires help now and then.

A number of design issues must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction?
- How will the user request help?
- How will help be represented?

Error handling – Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration.

In general, every error message or warning produced by an interactive system should have the following characteristics:

- The message should describe the problem in jargon that the user can understand.
- The message should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error

Menu and command labeling – The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type.

Application accessibility – As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. Accessibility for users who may be physically challenged is an imperative for ethical, legal, and business reasons.

Internationalization – The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software.

3.12. COMPONENT LEVEL DESIGN

Component level design occurs after the first iteration of architectural design has been completed. A component is a modular building block for computer software. More formally, the OMG Unified Modeling Language specification defines a component as a modular, deployable and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.

3.12.1. AN OBJECT-ORIENTED VIEW

In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. Example illustrates the process of design elaboration is print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility.

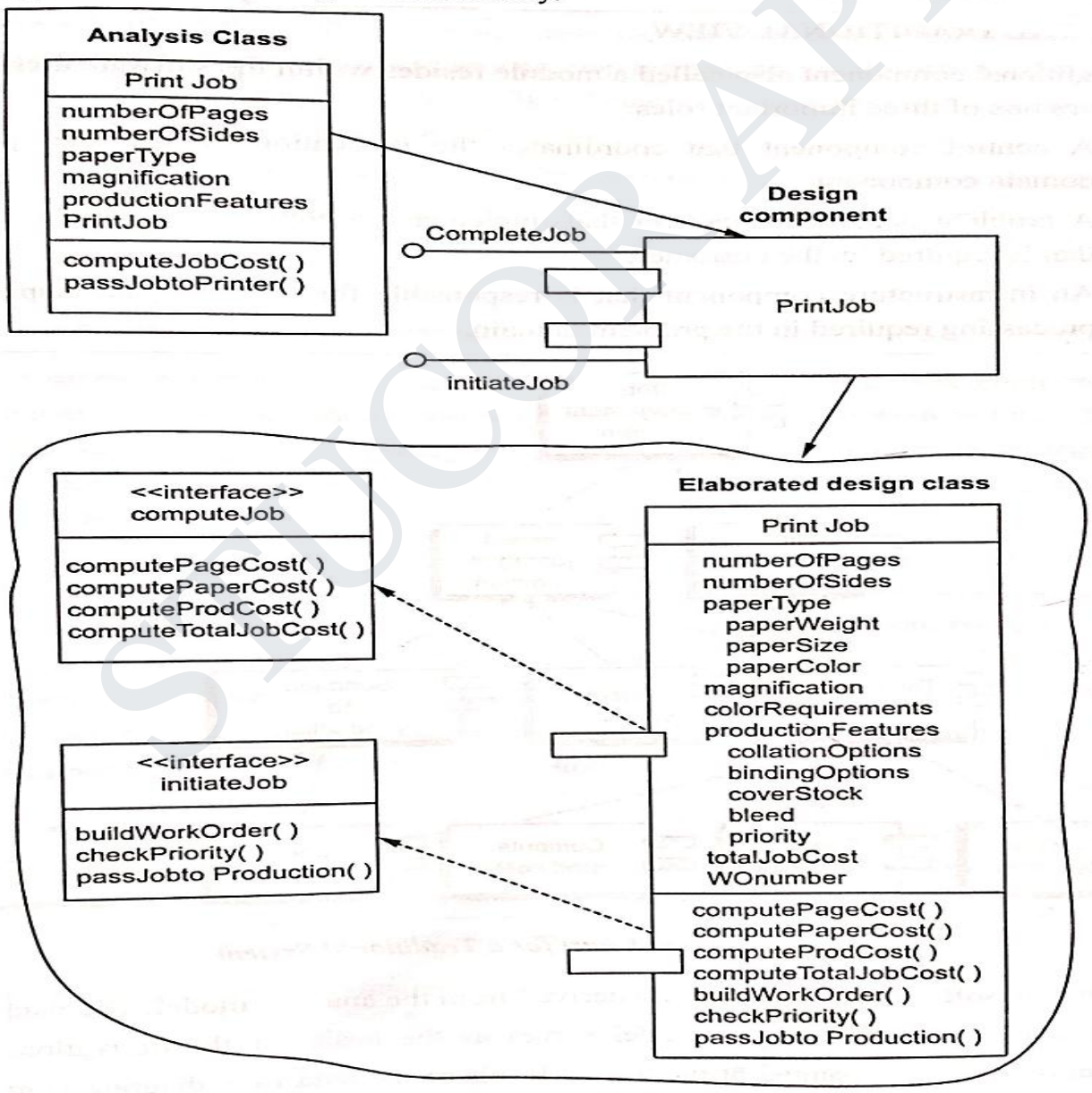


Fig. 3.29. Elaboration of a Design Component

3.12.2. THE TRADITIONAL VIEW

A traditional component also called a module resides within the software architecture and serves one of three important roles:

1. A control component that coordinates the invocation of all other problem domain components,
2. A problem domain component that implements a complete or partial function that is required by the customer,
3. An infrastructure component that is responsible for functions that support the processing required in the problem domain.

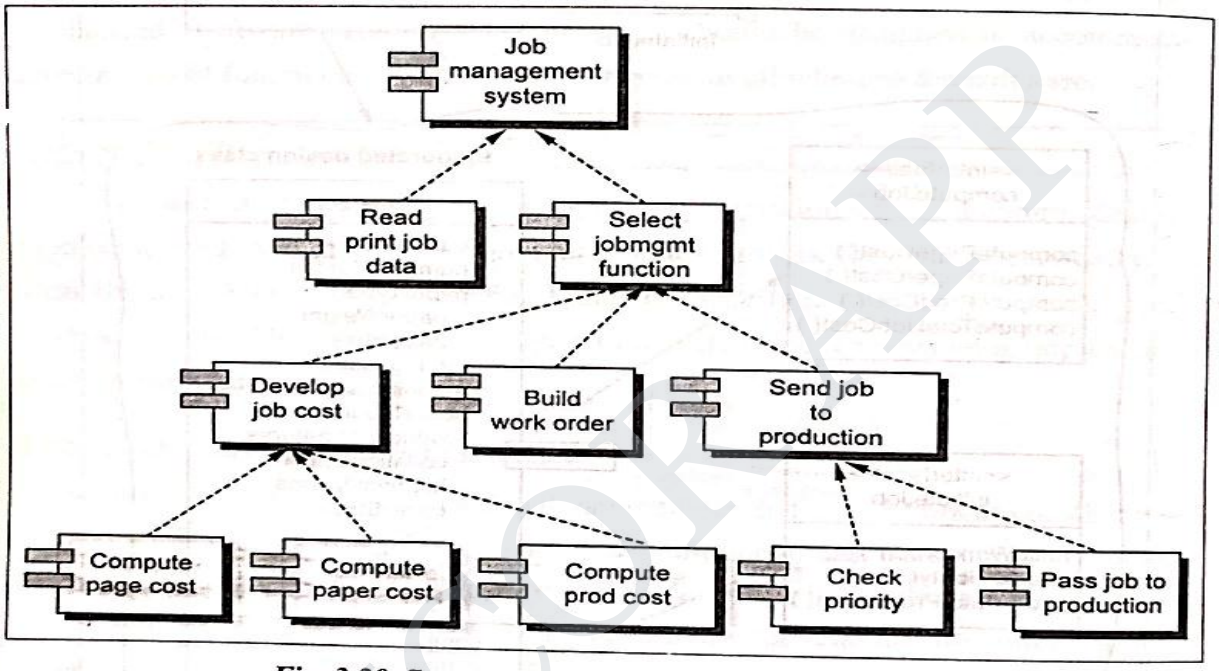


Fig. 3.30. Structure Chart for a Traditional System

Traditional software components are derived from the analysis model. The data flow-oriented element of the analysis model serves as the basis for the derivation. Each transform (bubble) represented at the lowest levels of the data flow diagram is mapped into a module hierarchy.

Control components (modules) reside near the top of the hierarchy and problem domain components tend to reside toward the bottom of the hierarchy. To achieve effective modularity, design concepts like functional independence are applied as components are elaborated. Example is print shop.

A set of data flow diagrams would be derived during requirements modeling. Assume that these are mapped into an architecture. Each box represents a software component. The shaded boxes are equivalent in function to the operations defined for the PrintJob class.

3.12.3. A PROCESS-RELATED VIEW

As the software architecture is developed, software engineers choose components or design patterns from the catalog and use them to populate the architecture. Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available to software engineers.

3.13. DESIGNING CLASS-BASED COMPONENTS

When an object-oriented software engineering approach is chosen, component level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure class contained in the requirements model. The detailed description of the attributes, operations and interfaces used by the classes in the design detail required as a precursor to the construction activity.

3.13.1. BASIC DESIGN PRINCIPLES

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The principles are used here as a guide as each software component is developed.

3.13.1.1. The Open-Closed Principle (OCP)

A module (component) should be opened for extension but closed for modification. This statement seems to be a contradiction but it represents one of the most important characteristics of a good complement-level design.

Example, Assume that the SafeHome Security function make use of a detector class that must check the status of each type of security sensor.

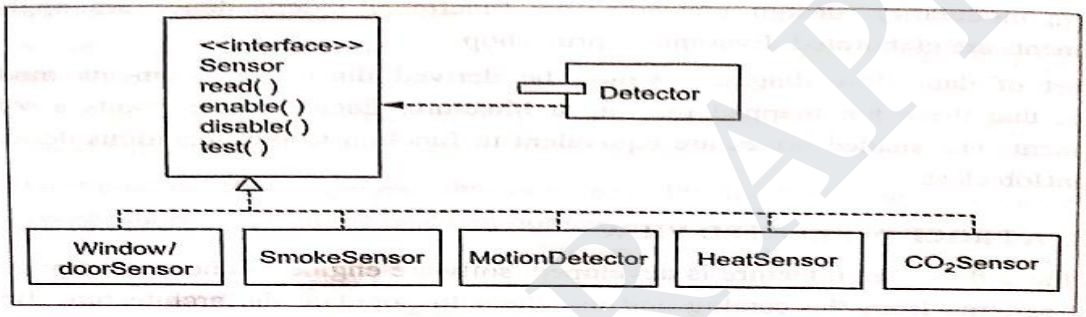


Fig. 3.31. Open Closed Principle

3.13.2. THE LISKOV SUBSTITUTION PRINCIPLE (LSP)

"Subclasses should be substitutable for their base classes". This design principle, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.

3.13.3. DEPENDENCY INVERSION PRINCIPLE (DIP)

"Depend on abstractions. Do not depend on concretions". The more a component depends on other concrete components, the more difficult it will be to extend.

3.13.4. THE INTERFACE SEGREGATION PRINCIPLE (ISP)

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

Additional packaging principles are applicable to component-level design.

3.13.5. THE RELEASE REUSE EQUIVALENCY PRINCIPLE (REP)

"The granule of reuse is the granule of release". When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it.

3.13.6. THE COMMON CLOSURE PRINCIPLE (CCP)

"Classes that change together belong together.". Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioural area.

3.13.7. THE COMMON REUSE PRINCIPLE (CRP)

"Classes that aren't reused together should not be grouped together". When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident.

3.13.8. COMPONENT-LEVEL DESIGN GUIDELINES

Guidelines can be applied as component-level design proceeds. These guidelines apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design.

Components – Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Example, Floor Plan – Class Name.

Interfaces – Interfaces provide important information about communication and collaboration.

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

3.13.11. COHESION

Cohesion is of a module represents how lightly bound the internal elements of the module are to one another. It is a natural extension of information hiding and it is the measure of the relative functional strength of a module. There are several levels of cohesion.

Coincidental Cohesion

Coincidental cohesion occurs when the elements within a module have no apparent relationship to one another. It occurs when an existing program is modularized into prices and making different pieces modules.

Logical Cohesion

A module has logical cohesion if there is some logical relationship between the elements of the module and the elements perform functions that fall in the same logical class.

Temporal cohesion

Temporal cohesion is the same as logical cohesion, except that the elements are also related in time and are executed together. Modules that perform activities like "initialization," "cleanup," and "termination" are usually temporally bound.

Modules with temporal cohesion exhibit many of the same disadvantage as logically bound modules. However, they are higher on the scale of binding because all elements are executed at one time, and no parameters or logic are required to determine which elements to execute.

Procedural Cohesion

A procedurally cohesion module contains elements that belong to a common procedural unit. A module with only procedural cohesion may contain only part of a complete function or parts of several functions.

Communicational Cohesion

A module with communicational cohesion has elements that are related by a reference to the same input or output data. Communicational binding is higher on the binding scale than temporal binding because the elements are executed at one time and also refer to the same data.

Sequential Cohesion

Sequential cohesion of elements occurs when the output of one element is the input for the next element. Sequential cohesion is high on the binding scale because the modular structure usually bears a close resemblance to the problem structure.

Functional Cohesion

Functional Cohesion is the strongest cohesion. In a functionally bound module, all the elements of the module are related to perform a single function. Modules with functional cohesion can always be described by a simple structure but sometimes it can also be described using compound sentences.

Cohesion Level	cohesion attribute	resultant module strength
Coincidental	low cohesion	weakest
Logical		
Temporal		
Procedural		
Communicational		
Sequential		
Functional	high cohesion	strongest

3.13.12. COUPLING

Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module and what data path across the interface.

"Highly coupled" modules are joined by strong interconnection while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections.

Content Coupling

Content Coupling occurs when one module modifies local data or instructions in another modules. Content coupling occurs when branches are made into the middle of the module.

Common Coupling

In common coupling, modules are bound together by global data structures. While common coupling it can be difficult to determine which component is responsible for having set of variable for a particular value.

Control Coupling

When one component passes parameters to control the activity of another component, we say that there is control coupling between the two. In a design with control coupling, there is an advantage to have each component perform only on functions or execute one process. It involves passing of control flags such as parameters or globals between modules.

Stamp Coupling

Stamp coupling is similar to common coupling except that global data items are shared selectively among routines that require the data. Stamp Coupling is more desirable than common coupling because fewer modules will have to be modified if a shared data structure is modified.

Data Coupling

Data coupling involves the use of parameter lists to pass data items between components. If coupling must exist between components, data coupling is the most desirable, it is the easiest to through which to trace data and to make changes.

Routine Call Coupling

Occurs when one operation involves another. This level of coupling is common and is often quite necessary. However it does increase the connectedness of a system.

Type use Coupling

Occurs when component A uses a data type defined in component B. If the type definition changes, every component that uses the definition must also change.

Inclusion or Import Coupling

Occurs when component A imports or includes a package or the content of component B.

External Coupling

Occurs when a component communicates or collaborates with infrastructure components (Example, Operating system function, database capability, telecommunication functions).

Coupling	Coupling Attribute	Resultant Module Strength
Content coupling	High coupling	Weakest
Common coupling		
Control coupling		
Stamp coupling		
Data coupling	Low coupling	strongest

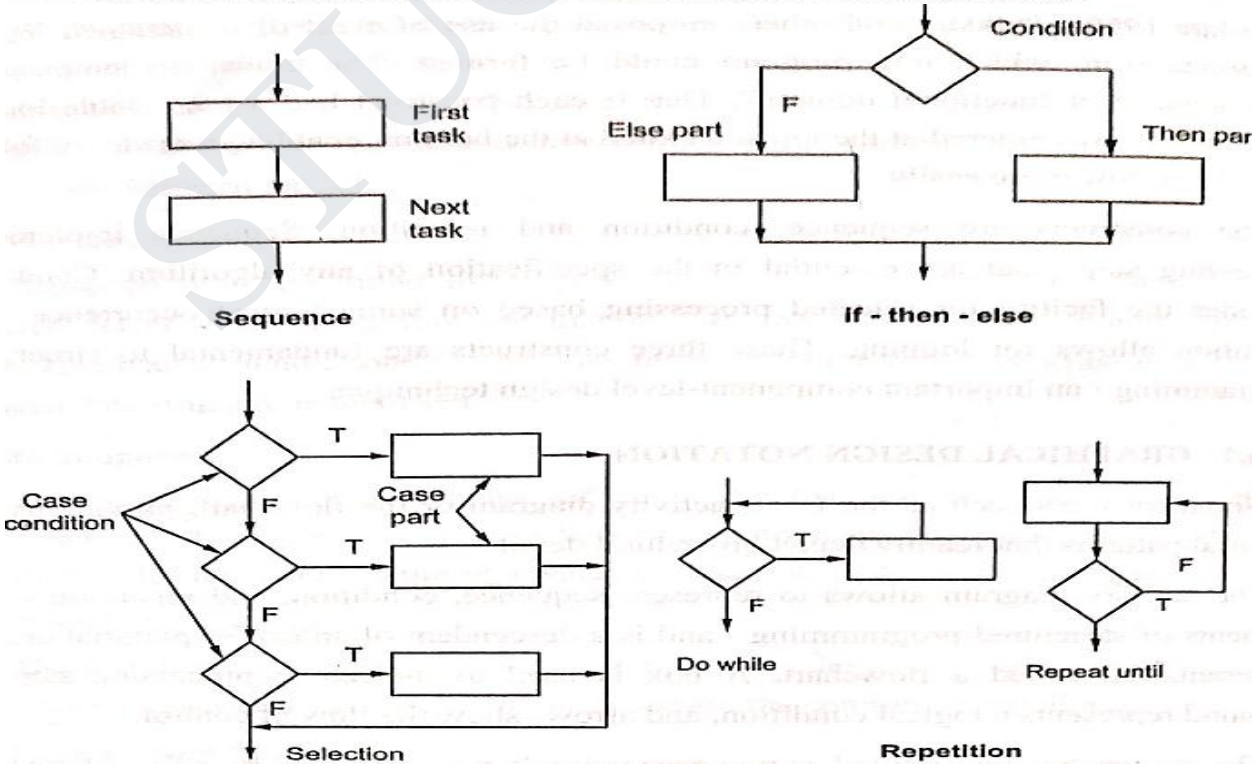
3.14 DESIGNING TRADITIONAL COMPONENTS

In late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasised "maintenance of functional domain". That is each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

3.14.1. GRAPHICAL DESIGN NOTATION

Graphical tools such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail.

The activity diagram allows to represent sequence, condition, and repetition - all elements of structured programming - and is a descendent of an earlier pictorial design representation called a flowchart. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control.



The sequence is represented as two processing boxes connected by a line of control. Condition, also called if-then-else, is depicted as a decision diamond that, if true, causes then-part processing to occur, and if false, invokes else-part processing.

Repetition is represented using two slightly different forms. The do while tests a condition and executes a loop task repetitively as long as the condition holds true.

A repeat until executes the loop task first and then tests a condition and repeats the task until the condition fails. The selection (or select-case) construct is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

3.14.2. TABULAR DESIGN NOTATION

Decision tables provide a notation that translates actions and conditions into a tabular form.

Decision table is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a marix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing rule.

Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

Fig. 3.33. Decision Table Nomenclature

3.14.3. PROGRAM DESIGN LANGUAGE

Program design language (PDL), also called structured English or pseudocode, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English).

A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs.

Example of PDL, Consider a procedural design for the SafeHome security function:

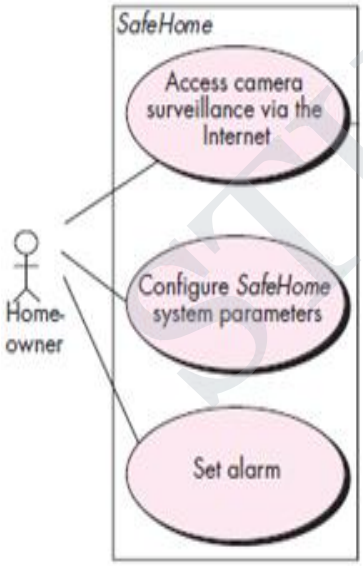
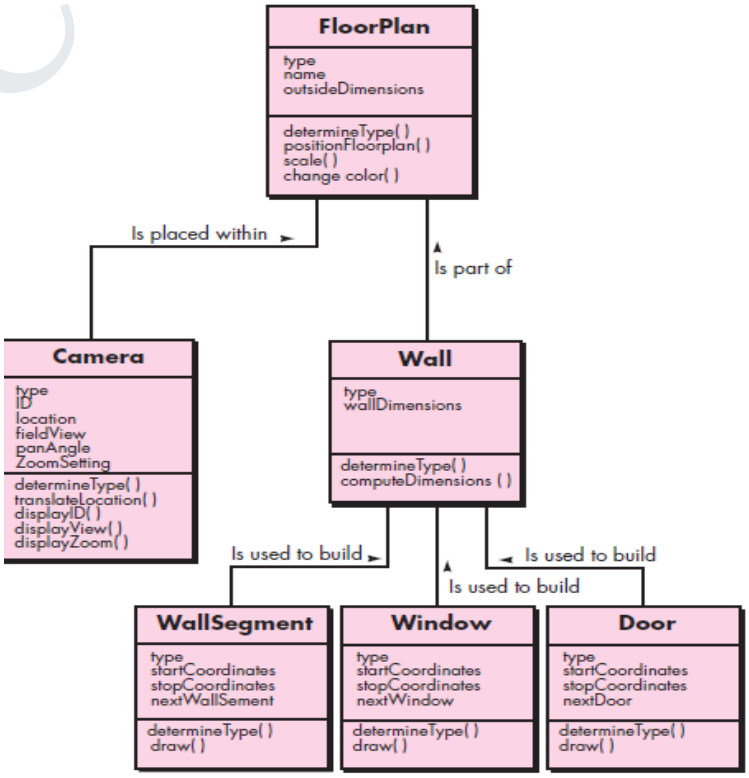
```
component alarmManagement;
The intent of this component is to manage control panel switches and input from sensors by
type and to act on any alarm condition that is encountered.
    set default values for systemStatus (returned value), all data items
    initialize all system ports and reset all hardware
```

```
check controlPanelSwitches (cps)
    if cps = "test" then invoke alarm set to "on"
    if cps = "alarmOff" then invoke alarm set to "off"
    if cps = "newBoundingValue" then invoke keyboardInput
    if cps = "burglarAlarmOff" invoke deactivateAlarm;
    •
    •
    •
    default for cps = none
reset all signalValues and switches
do for all sensors
    invoke checkSensor procedure returning signalValue
    if signalValue > bound [alarmType]
        then phoneMessage = message [alarmType]
            set alarmBell to "on" for alarmTimeSeconds
            set system status = "alarmCondition"
            parbegin
                invoke alarm procedure with "on", alarmTimeSeconds;
                invoke phone procedure set to alarmType, phoneNumber
            endpar
        else skip
    endif
enddofor
end alarmManagement
```

Program Design Language(PDL)

The construct parbegin … parend that specifies a **parallel block**. All tasks specified within the parbegin block are executed in parallel.

Differences between object oriented and functional oriented design:

Functional Oriented Design	Object Oriented Design
Begins by considering the **use case diagrms ,Scenarios**	Begins by identifying **objects and classes**
	

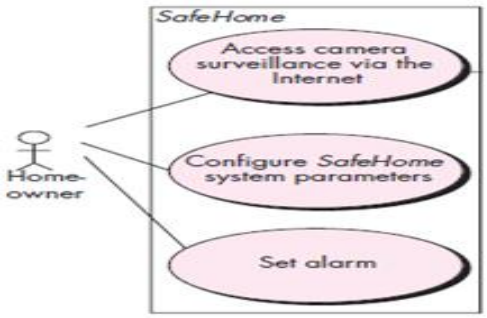
Decompose in **function/procedure level**	Decompose in **class level**
Top down Approach	Bottom up approach
Performs high level **function** and later decompose it **detailed function**	Defining the **objects** and their **interactions** to solve a problem
The state information is often represented in a **centralized shared memory**.	The state information is not represented in a centralized memory but is implemented or **distributed** among the **objects** of the system
Used for **computation sensitive application**	Used for evolving system which mimicks a **business process.**

SAFE HOME SECURITY SYSTEM

SCENARIO FOR SAFE HOME SECURITY SYSTEM

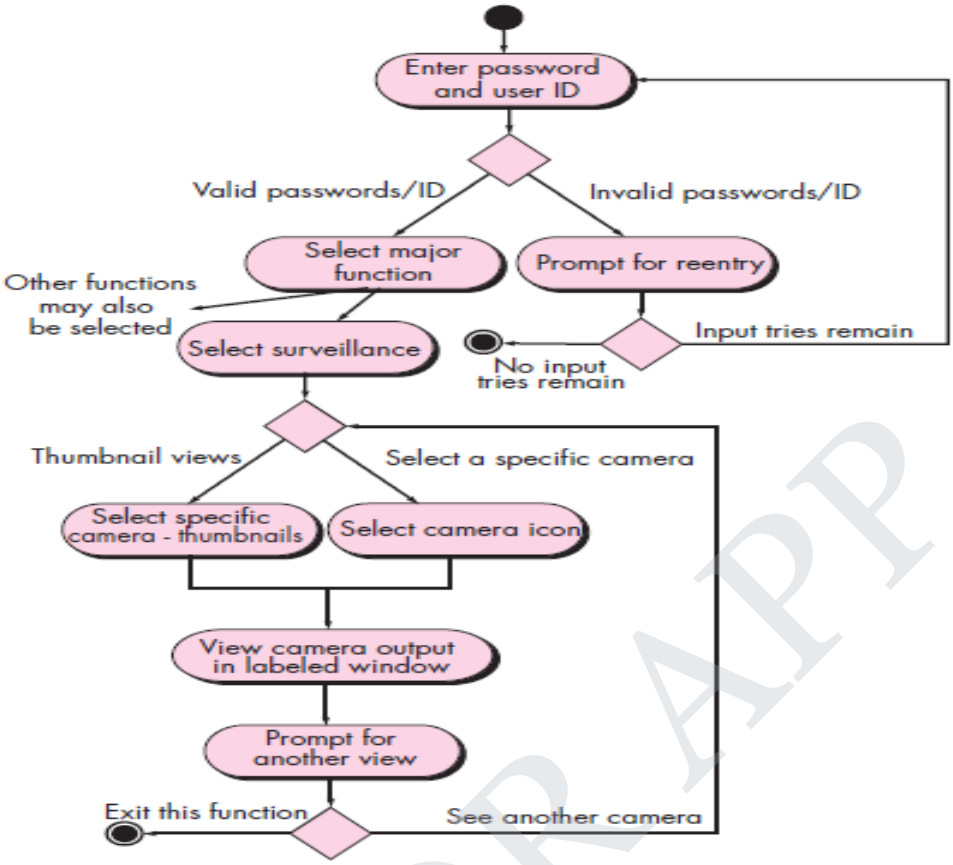
Primary actor:	Homeowner.
Goal:	**To view output of camera placed** throughout the house from any remote location via the Internet.
Preconditions	**System must be fully configured;** appropriate user ID and passwords must be obtained.
Trigger	**The homeowner decides to take** a look inside the house while away
Action:	1. The homeowner logs onto the *SafeHome Products* website. 2. The homeowner enters his or her user ID. 3. The homeowner enters two passwords (each at least eight characters in length). 4. The system displays all major function buttons. 5. The homeowner selects the "surveillance" from the major function buttons. 6.The homeowner selects "pick a camera." 7. The system displays the floor plan of the house. 8. The homeowner selects a camera icon from the floor plan. 9. The homeowner selects the "view" button. 10. The system displays a viewing window that is identified by the camera ID. 11. The system displays video output within the viewing window at one frame per second.
Exceptions:	1. ID or passwords are incorrect or not recognized 2. Surveillance function not configured. 3. A floor plan is not available or has not been configured

Usecase Diagram for Safe Home Security System

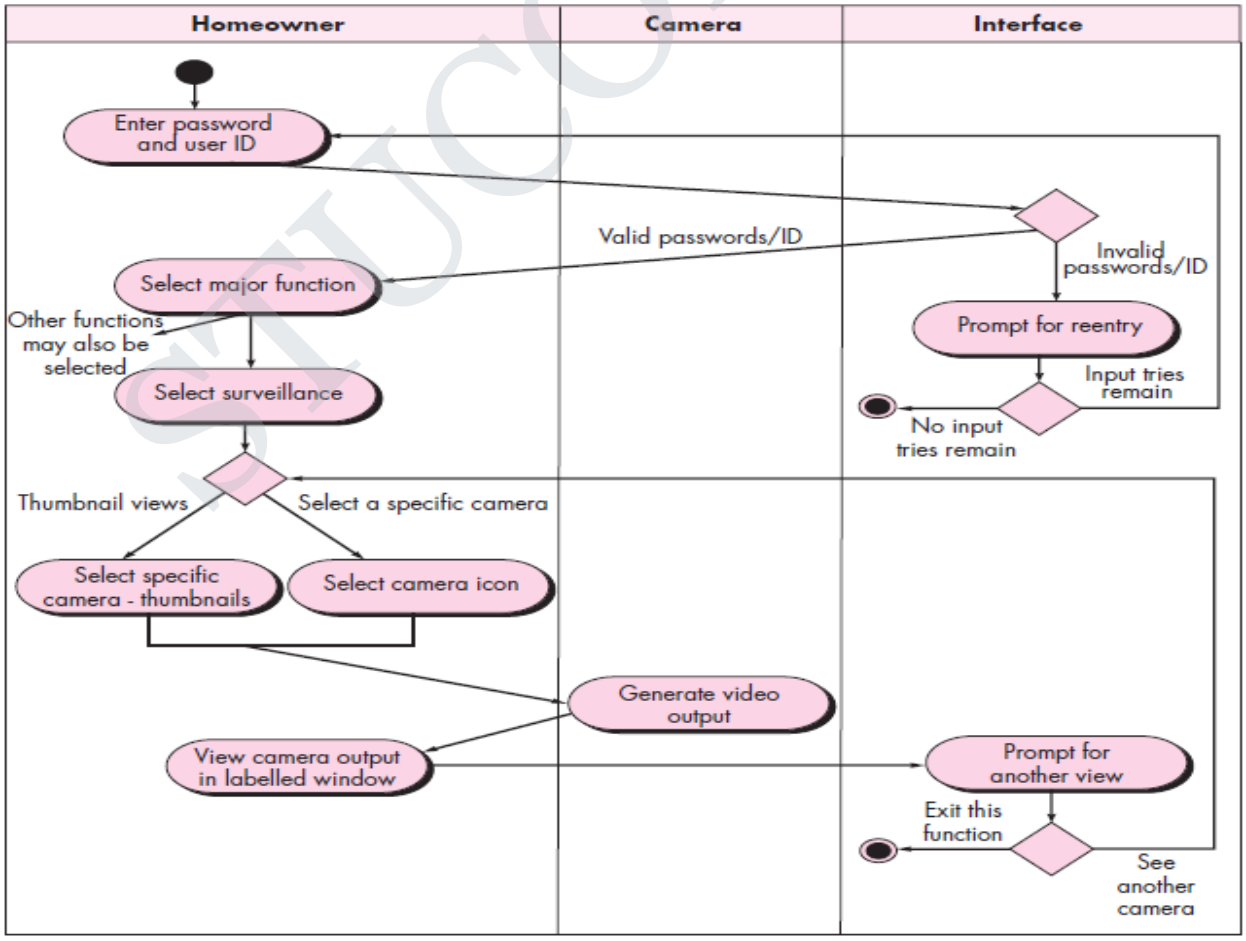


use case—may not impart information in a clear manner. In such cases, you can choose from broad array of **UML graphical models.**

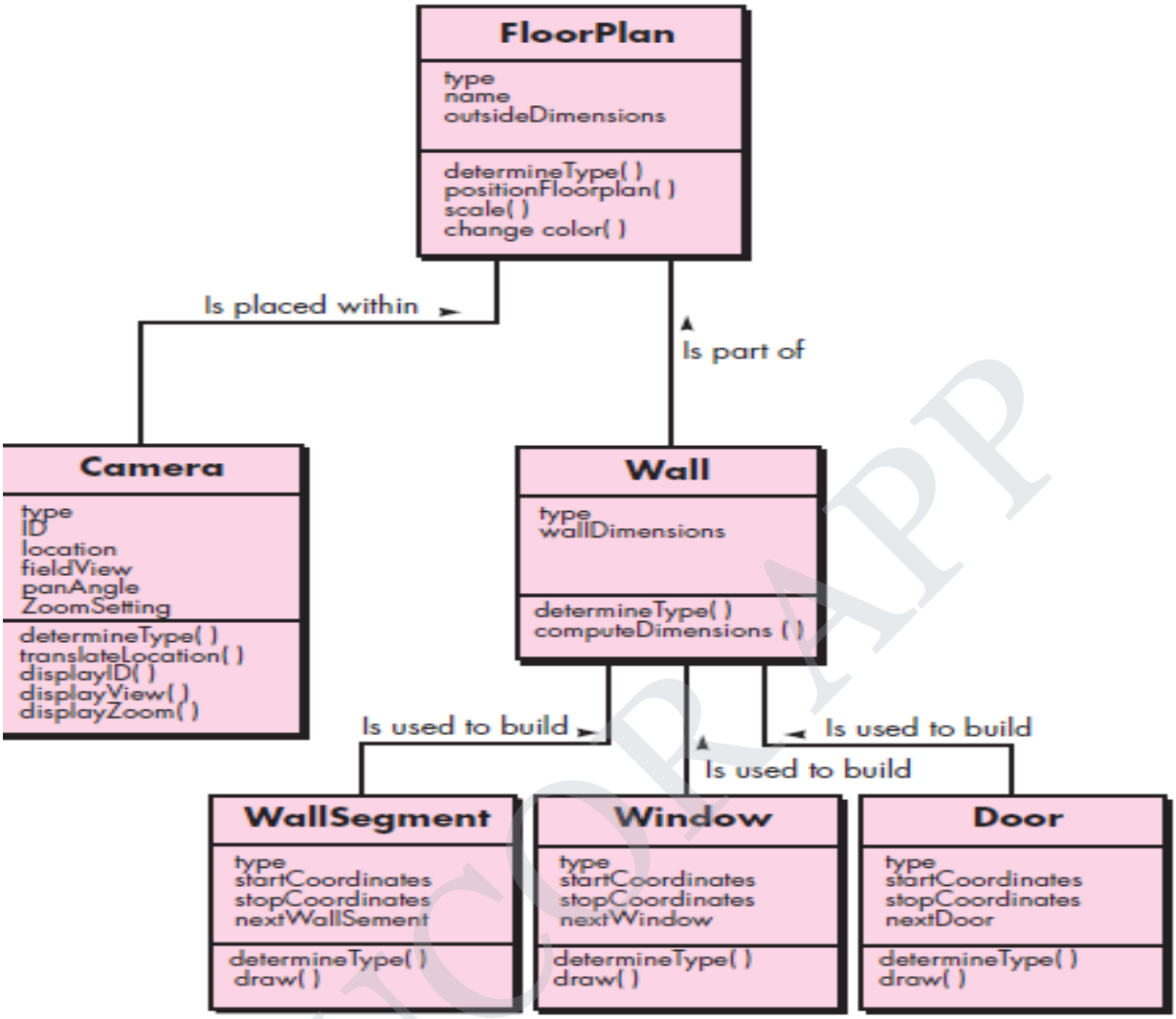
ACTIVITY DIAGRAM for Safe Home Security System



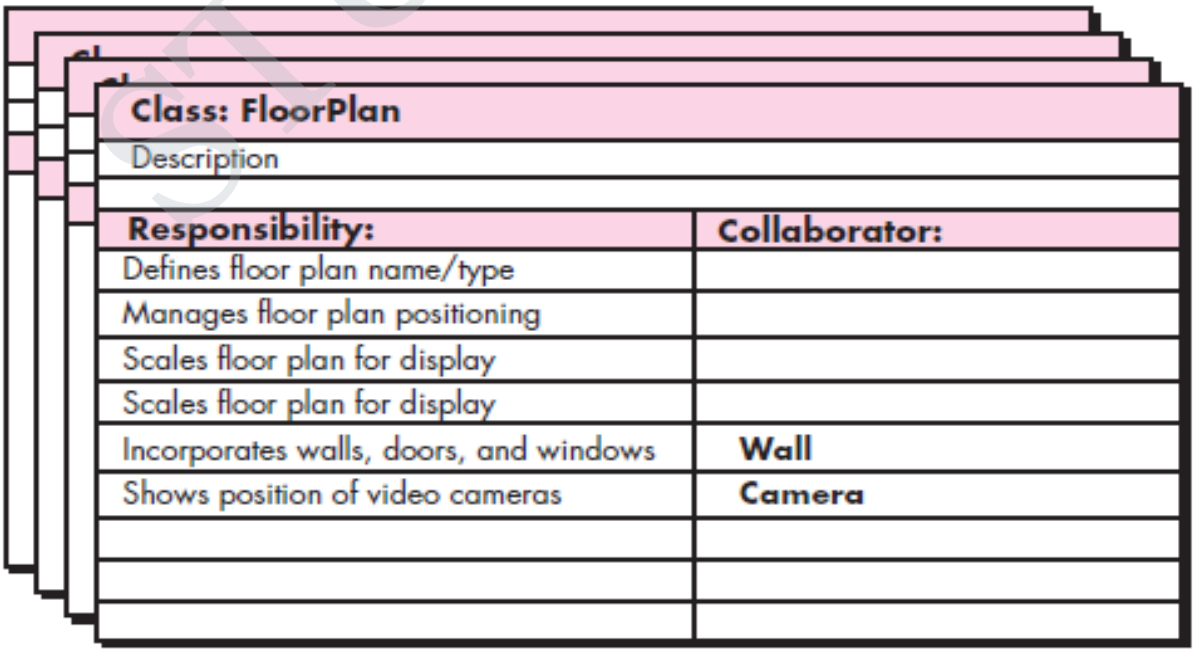
SWIMELANE DIAGRAM for Safe Home Security System



CLASS DIAGRAM for floorplan

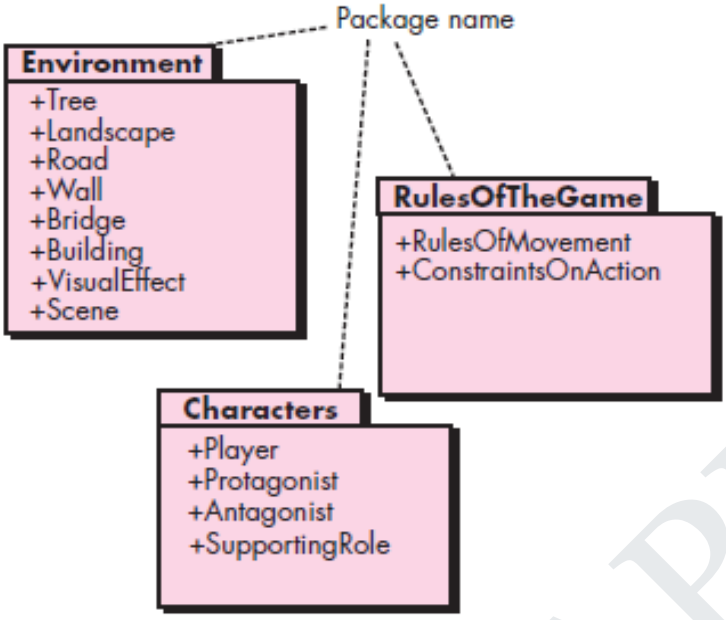


CRC model for floorplan



Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

PACKAGE DIAGRAM for video game



UNIVERSITY QUESTIONS
What is system modeling? Explain the process of creating models and the factors that should be considered when building models. **(NOV/DEC 2013)**
Explain the core activities involved in user interface design process with necessary block diagrams. **(NOV/DEC 2010) (NOV/DEC 2016)**
(i) Discuss the design heuristics for effective modularity design.**(NOV/DEC 2016)** (ii)Explain the architectural styles used in the architectural design. **(MAY/JUNE 2013, 2014)**
Discuss class-based components along with its principles in detail. For a Case study of your choice show the architectural and Component design. **(APR/MAY 2015)**
Discuss about software architecture design, which emphasize on fan in, fan out, coupling, cohesion and factoring? **(NOV/DEC 2011) (NOV/DEC 2012)(APR/MAY 2015)**
Explain the basic concepts of software design. **(APR/MAY 2011)(NOV/DEC 2014)**
Discuss the process of translating the analysis model into a software design. **(APR/MAY 2011)**
i) What is modularity? state its importance and explain coupling and cohesion**. (MAY/JUNE 2016)** ii) Discuss the differences between object oriented and functional oriented design. **(MAY/JUNE 2016)**
i) Describe the golden rules for interface Design. **(NOV/DEC 2016)** ii) Explain Component level design with suitable examples**. (NOV/DEC 2016)**
What is Software Architecture? Describe in detail different types of software architecture with illustrations. **(APR/MAY 2017)**

UNIT IV TESTINGANDIMPLEMENTATION

Software testing fundamentals-Internal and external views of Testing-white box testing - basis path testing-control structure testing-black box testing- Regression Testing – Unit Testing – Integration Testing – Validation Testing – System Testing And Debugging – Software Implementation Techniques**:** Coding practices-Refactoring.

SOFTWARE TESTING FUNDAMENTALS

A good test must achieve the goal of finding the maximum errors with minimum effort.

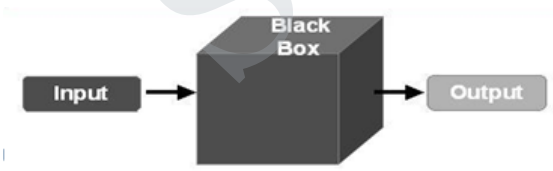
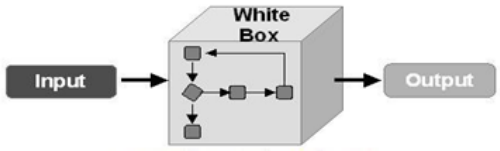
TESTABILITY / CHARACTERISTICS OF TESTABLE SOFTWARE

Operability	Allowing testing to progress without fits and starts
Observability	Visibility of system states and variables
Controllability	Software, hardware states and variables can be controlled directly by test engineer
Decomposability	Isolate problems and perform smarter testing
Simplicity	• Functional simplicity (minimum feature set) • Structural simplicity (modularized architecture) • Code simplicity (standard coding ease of inspection and maintenance)
Stability	Changes made to the software are infrequent- fewer disruptions to testing
Understandability	System is well-understood, technical documentation is well-organized and accurate

TEST CHARACTERISTICS

- A good test has a high probability of finding an error
- A good test is not redundant
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex

INTERNAL AND EXTERNAL VIEWS OF TESTING

EXTERNAL VIEW OF TESTING	INTERNAL VIEW OF TESTING
1. External view is called ***black-box testing***	***Internal view*** is termed ***white-box testing***
2. Black-box testing is also known as ***behavioral testing***	White-box testing is known as ***glass-box testing***
3. Tests are conducted at the software ***interface***	Test cases exercise ***program logic*** exhaustively
4. Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT knownto the tester	White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
5. Tests are conducted that demonstrate each ***function is fully operational***	***Control structures*** are tested for validity
6. 	
7. Graph-based testing methods, equivalence partitioning, boundary value analysis and orthogonal array testing are a few black-box techniques.	Basis path testing, condition testing, data flow testing and loop testing are a few white-box techniques.
8. Mainly applicable to higher levels oftesting: •Acceptance Testing •System Testing	Mainly applicable to lower levelsof testing: • Unit Testing •Integration Testing
9. Responsibilityof independent Software Testers	Responsibility of software developers generally
10. Programming knowledge and implementation knowledge not required.	Programming knowledge and implementation knowledge not required.

11.Black-box testing attempts to find errors in 1. incorrect or missing functions 2. interface errors 3. errors in data structures or external database access 4. behavior or performance errors 5. initialization and termination errors.	Using white-box testing methods, derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once (2) exercise all logical decisions on their true and false sides (3) executeall loops at their boundaries and within their operational bounds and (4) exerciseinternal data structures to ensure their validity.

WHITE BOX TESTING

White-box testingcalled ***glass-box testinguses the control structure*** described as part of component-level design to derive test cases. Test cases

- 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
- exercise internal data structures to ensure their validity
-

BASIS PATH TESTING

Basis path testing is a white-box testing technique first proposed by ***Tom McCabe***. 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***. Test cases derived to exercise the basis set are guaranteed to ***execute every statement in the program at least one time during testing***.

Steps in Basis Path Testing- Deriving Test Cases:

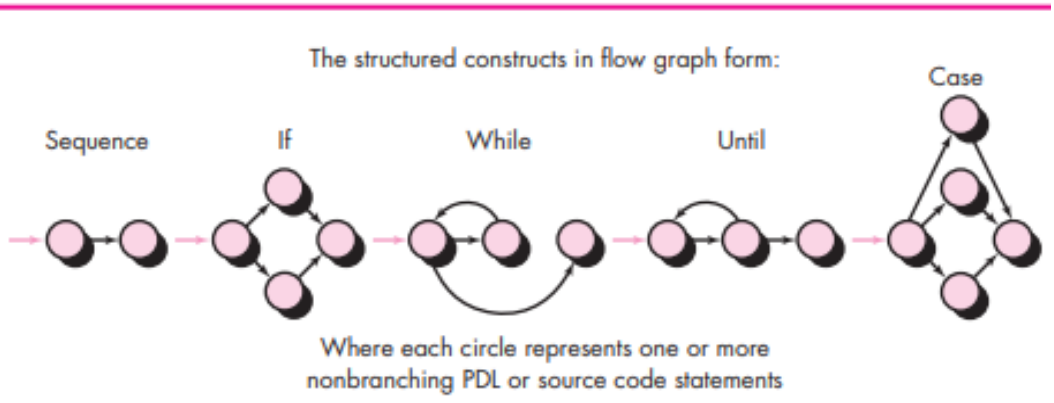
1. Using the design or code as a foundation, draw a correspondingflow graph.
2. Determine the cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basisset.

Flow Graph Notation

The flow graph depicts logical control flow using the notation. A flow graph should be drawn only when the logical structure of a component is complex. The flow graph allowsto trace program paths more readily.

Notation	Explanation
Circle	Flow graph node (one or more procedural statements)
Arrows	Edges/Links (Flow of control)

FIGURE 18.1
Flow graph notation



Region- Areas bounded by edges and nodes
Predicate Node- Node that contains a condition and two/more edges emanating from it

FIGURE 18.2 (a) Flowchart and (b) flow graph

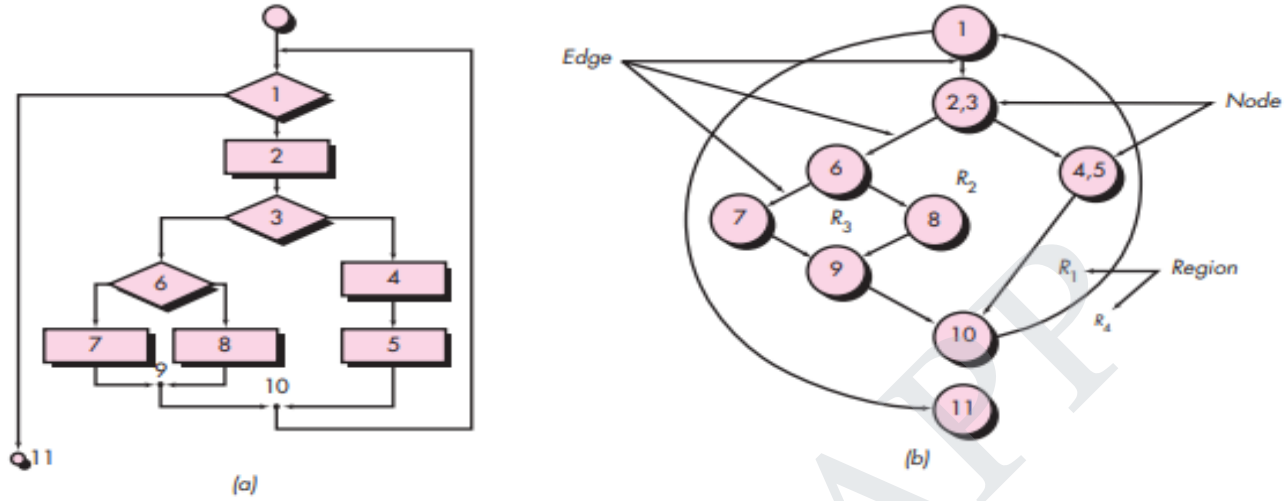
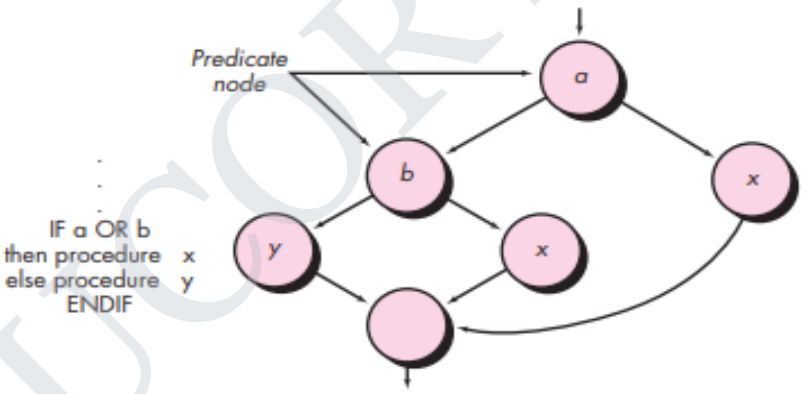


FIGURE 18.3 Compound logic



Independent Program Paths

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
Basis set is the set of all independent paths.
For example, a set of independent paths for the flow graph illustrated in Figure 18.2b is

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

Basis set

Each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges. The basis set is not unique for a given procedural design.**On executing all paths mentioned in the basis set, each statement in the program is guaranteed to be executed at least once.**

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.For a flow graph G,Cyclomatic complexity V(G) is computed in one of three ways:

Method 1:V(G) = Number of regions of the flow graph
In the example, the graph has four regions.

Method 2:V(G) =E-N+ 2 where E is the number of flow graph edges, N is the number of flow graph nodes.
In the example, V(G)= 11 edges – 9 nodes +2 = 4

Method 3:V(G)=P+1 where P is the number of predicate nodes contained in the flow graph G.
In the example, V(G) = 3 predicate nodes+1 = 4

Therefore, the cyclomatic complexity of the flow graph is 4.

Graph Matrices/ Procedure for deriving flow graph / Data Structure of flow graph:

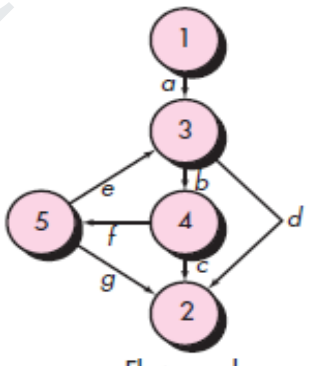
A data structure, called a graph matrix, can be useful for developing ***a software tool that assists in basis path testing***.A graph matrix is a ***square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph***. Each row and column correspondsto an identified node, and matrix entries correspond to connections (an edge)between nodes.The graph matrix is nothing more than ***a tabular representation of a flow graph.*** However, by adding a link weight to each matrix entry, the graph matrix can become a powerful ***tool for evaluating program control structure during testing***.The link weight provides additional information about control flow. The link weight 1 indicates a connection exists or 0 indicates a connection does not exist.

Properties of link **weights** are:

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

Beizer provides a thorough treatment of additional mathematical algorithmsthat can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

FIGURE 18.6
Graph matrix



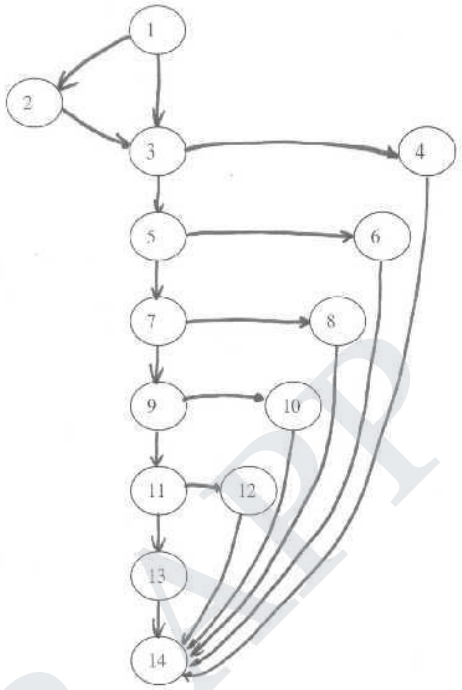
Flow graph

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

Graph matrix

Basis Path Testing Example
Step 1: Draw the flow graph for the algorithm.

The example procedure below shows how the algorithm statements are mapped into graph nodes, numbered on the left.

public double calculate(int amount)
{
-1- double rushCharge = 0;
-1- if (nextday.equals("yes"))
{
-2- rushCharge = 14.50;
}
-3- double tax = amount * .0725;
-3- if (amount >= 1000)
{
-4- shipcharge = amount * .06 + rushCharge;
}
-5- else if (amount >= 200)
{
-6- shipcharge = amount * .08 + rushCharge;
}
-7- else if (amount >= 100)
{
-8- shipcharge = 13.25 + rushCharge;
}
-9- else if (amount >= 50)
{
-10- shipcharge = 9.95 + rushCharge;
}
-11- else if (amount >= 25)
{
-12- shipcharge = 7.25 + rushCharge;
}
else
{
-13- shipcharge = 5.25 + rushCharge;
}
-14- total = amount + tax + shipcharge;
-14- return total;
} //end calculate
	Here is a drawing of the flowgraph.
	 

Step 2: Determine the **cyclomatic complexity** of the flow graph.

V(G) = E - N + 2 = 19 - 14 + 2 = 7

V(G)=P+1 = 6+1= 7

V(G)=number of region =7

This tells us the *upper bound* on the size of the basis set. That is, it gives us the number of independent paths we need to find.

Step 3: Determine the basis **set of independent paths.**

Path 1: 1 - 2 - 3 - 5 - 7 - 9 - 11 - 13 - 14

Path 2: 1 - 3 - 4 - 14

Path 3: 1 - 3 - 5 - 6 - 14

Path 4: 1 - 3 - 5 - 7 - 8 - 14

Path 5: 1 - 3 - 5 - 7 - 9 - 10 - 14

Path 6: 1 - 3 - 5 - 7 - 9 - 11 - 12 - 14

Path 7: 1 - 3 - 5 - 7 - 9 - 11 - 13 - 14

Note: This basis set is not unique. There are several different basis sets for the given algorithm. You may have derived a different basis set.

The basis set "covers" all the nodes and edges in the algorithm.

Step 4: **Prepare test cases** that force execution of each path in the basis set.

path	nextday	amount	expected result
1	yes	10	30.48
2	no	1500	1698.75
3	no	300	345.75
4	no	150	174.125
5	no	75	90.3875
6	no	30	39.425
7	no	10	15.975

CONTROL STRUCTURE TESTING

1. Condition Testing:

Errors are much morecommon in theneighborhood oflogical conditions thanthey are in the locusofsequential processingstatements.Condition testing is a test-case design method that exercises the logical conditionscontained in a program module.

CONDITION TYPE	INCLUDES
Simple condition	1. Boolean variable 2. Relational expression of the form E1 <relational-operator> E2 where E1 and E2 are arithmetic expressions and <relational-operator> can be $<, \leq, =, \neq$ (nonequality), $>$, or $\geq$.
Compound condition	1. Two or more simple conditions 2. Boolean operators OR (|), AND (&) and NOT $(\neg)$. 3. Parenthesis

If a condition is incorrect, then at least one component of the condition is incorrect.Therefore, **types of errors** in a condition include Boolean operator errors(incorrect/missing/extra Boolean operators), Boolean variable errors, Booleanparenthesis errors, relational operator errors, and arithmetic expression errors. Thecondition testing method focuses on testing each condition in the program to ensurethat it does not contain errors.

2. 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.

Assumptions:

1. Each statement in a program is assigned a unique statement number
2. Each function does not modify its parameters or global variables

For a statement with S as its statement number,

DEF(S) = {X | statement S contains a definition of X}

USE(S) = {X | statement S contains a use of X}

If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said tobe live at statement S' if there exists a path from statement S to statement S' that containsno other definition of X.

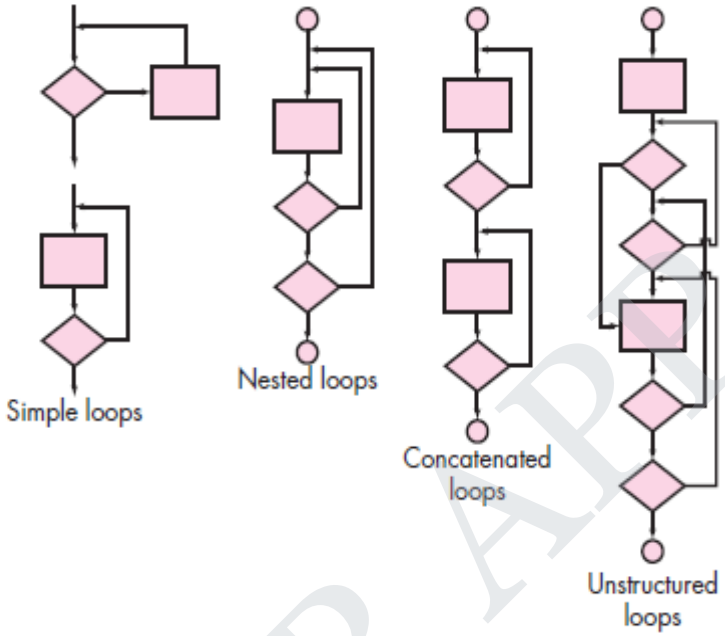
A **definition-use** (DU) chain of variable X is of the form [X, S, S'], where S and S' arestatement numbers, X is in DEF(S) and USE(S'), and the definition of X in statementS is live at statement S'.

One simple data flow testing strategy is to require that every DU chain be coveredat least once. We refer to this strategy as the DU testing strategy. It has been shownthat DU testing does not guarantee the coverage of all branches of a program in rare situationssuch as if-then-else constructs in which the then part has no definition of any variableand the else part does not exist.

3. 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, and unstructured loops.

FIGURE 18.7
Classes of Loops



Simple loops:

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

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

Nested loops:

The number of possible tests would grow geometrically as the level of nestingincreases. This would result in an impractical number of tests. Beizer suggestsan approach that will help to reduce the number of tests:

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

Concatenated loops:

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops isindependent of the other. When the loops are not independent, the approach applied to nested loopsis recommended.

Unstructured loops:

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

BLACK BOX TESTING

Black-box testing, also called ***behavioral testing***, focuses on the ***functional requirements of the software***. Black-box testing is ***not an alternative to white-box techniques***. Rather, it is a complementary approach that is likely to uncover a different class of errors than white box methods.

Black box techniques derive a set of **minimum number of test cases** that achieve reasonable testing and indicate the presence or absence of **classes of errors**.

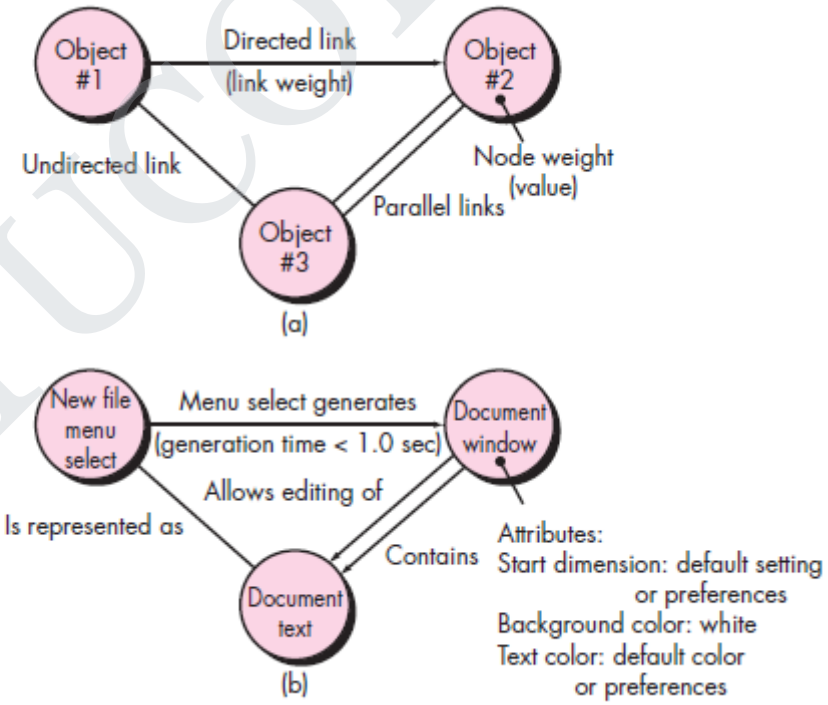
Graph-Based Testing Methods

Softwaretesting begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationshipis exercised and errors are uncovered.

TERMINOLOGY	INDICATED BY	DENOTES
Nodes	Circle	Objects
Links	Arrow	Relationship among objects **Directed link**- Relationship moves in only one direction **Undirected link**- Relationship applies in both directions **Parallel link**- number of different relationship established between graph nodes
Node weight	Specific data value/state behavior	Properties of node
Link weight	Specific data value/state transition condition	Characteristics of link

FIGURE 18.8

(a) Graph notation; (b) simple example



As a simple example, consider a portion of a graph for a word-processing application (Figure 18.8b) where

Object #1 newFile (menu selection)

Object #2 documentWindow

Object #3 documentText

Referring to the figure, a menu select on newFile generates a document window.The node weight of documentWindow provides a list of the window attributes thatare to be expected when the window is

generated. The link weight indicates that thewindow must be generated in less than 1.0 second. An undirected link establishes asymmetric relationship between the newFile menu selection and documentText, and parallel links indicate relationships between document Window and documentText. In reality, a far more detailed graph would have to be generatedas a precursor to test-case design. Derive test cases by traversing thegraph and covering each of the relationships shown. These test cases are designedin an attempt to find errors in any of the relationships.

Behavioral testing methods that can make use of graphs:

Transaction flow modeling: The nodes represent steps in some transactionand the links represent the logical connection between steps.

Finite state modeling: The nodes represent different user-observable states of the software, and the links represent the transitions that occur to movefrom state to state.

Data flow modeling: The nodes are data objects, and the links are thetransformations that occur to translate one data object into another.

Timing modeling: The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

Equivalence Partitioning:

Equivalence partitioning is a black-box testing method that **divides the input domain** of a program **into classes** of data from which test cases can be derived.An equivalence class represents a set of valid or invalid states for input conditions. Typically, an **input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition**.

Equivalence classes may be defined according to the following **guidelines:**

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

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

Boundary Value Analysis (BVA):

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." BVA extendsequivalencepartitioning by focusingon data at the "edges"of an equivalenceclass.Boundary value analysis is a test-case design technique that complements equivalence partitioning.

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

1. If an input condition specifies **a range** bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of **values**, test cases should be developed that exercise the minimum and maximum numbers. Values just aboveand below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.

For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have **prescribed boundaries** (e.g., a tablehas a defined limit of 100 entries), be certain to design a test case to exercisethe data structure at its boundary.Most software engineers intuitively perform BVA to some degree.

By applying these guidelines, boundary testing will be more complete, thereby having a higherlikelihood for error detection.

Equivalence and Boundary Value

- In Equivalence Partitioning, first divide a set of test conditions into partitions.
- In Boundary Value Analysis, test boundaries between equivalence partitions.

Example 1:

Suppose a password field accepts minimum 6 characters and maximum 10 characters.

Solution:

That means results for values in **partitions 0-5(invalid-EC1), 6-10(valid-EC2), 11-14(invalid-EC3) should be equivalent**.

Test Scenario #	Test Scenario Description	Expected Outcome
1	Enter 0 to 5 characters in password field	System should NOT accept
2	Enter 6 to 10 characters in password field	System should accept
3	Enter 11 to 14 character in password field	System should NOT accept

Evaluate using test cases:

TEST CASE	VALID EC	INVALID EC
4	**-**	**EC1**
7	**EC2**	**-**
11	**-**	**EC3**
13	**-**	**EC3**
8	**EC2**	

Example 2:

Suppose a Input Box field should accept the Number 1 to 10

Solution:

That means results for values in **partitions 0(invalid-EC1), 1-10(valid-EC2), >10(invalid-EC3) should be equivalent**.

Test Scenario	Test Scenario Description	Expected Outcome
1	Boundary Value = 0	System should NOT accept
2	Boundary Value = 1	System should accept
3	Boundary Value = 2	System should accept
4	Boundary Value = 9	System should accept
5	Boundary Value = 10	System should accept
6	Boundary Value = 11	System should NOT accept

Evaluate using test cases:

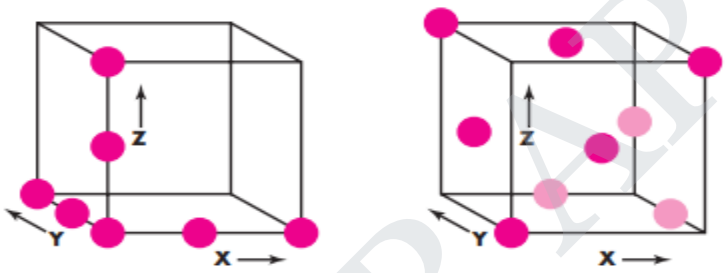
TEST CASE	VALID EC	INVALID EC
0	**-**	**EC1**
2	**EC2**	**-**
8	**EC2**	**-**
11	**-**	**EC3**
13	**-**	**EC3**

Orthogonal Array Testing:

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component.

Consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are 3^3=27 possible test cases. Phadke suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 18.9. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure). When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a "balancing property". That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure 18.9. Test coverage across the input domain is more complete.

FIGURE 18.9
A geometric view of test cases
Source: [Pha97]



One input item at a time — L9 orthogonal array

Consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values: P1 1, send it now P1 2, send it one hour later P1 3, send it after midnight P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), (1, 1, 1, 3) etc., There are 3^4=81 possible test cases.

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure 18.10.

FIGURE 18.10
An L9 orthogonal array

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

Phadke assesses the result of tests using the L9 orthogonal array in the following manner:

1. **Detect and isolate all "single mode faults":** A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1=1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 18.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1=1)] as the source of the error. Such an isolation of fault is important to fix the fault.
2. **Detect and isolate all "double mode faults":** If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault. Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.
3. **"Multimode faults"**: Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multimode faults are also detected by these tests.

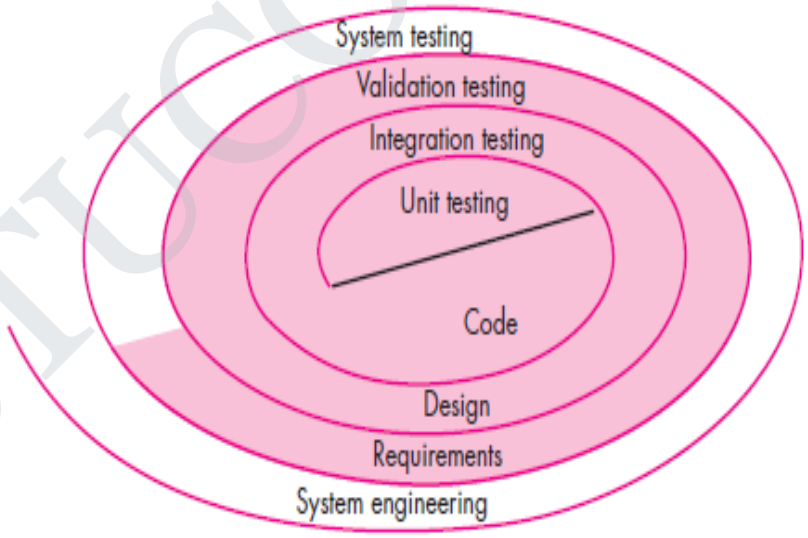
SOFTWARE TESTING STRATEGY_ THE BIG PICTURE

Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user. A good test must achieve the goal of finding the maximum errors with minimum effort.

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

FIGURE 17.1
Testing strategy



Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object)of the software as implemented in source code.)

Integration testing, where the focus is on design and the construction of the software architecture.

validation testing, where requirements established as part of requirements modeling are validated against the software that has been constructed.

Finally, you arrive at ***system testing***, where the software and other system elements are tested as a whole.

UNIT TESTING

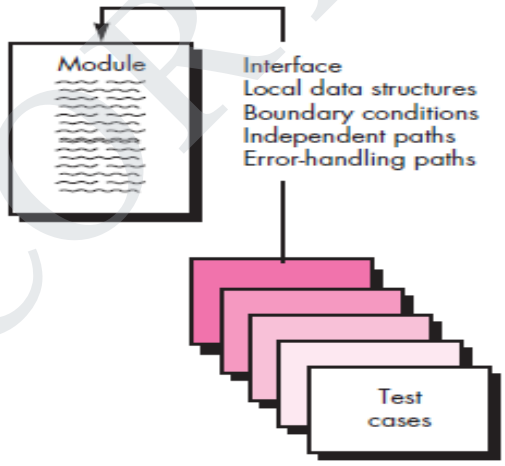
Testing of individual software components or modules. Method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use.

Each module tested individually with correct and in correct data

4.8.1. Unit Test Considerations

1. The module interface is tested to ensure that information properly flows into and out of the program unit under test.
2. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
3. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
4. 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.
5. And finally, all error handling paths are tested.

FIGURE 17.3
Unit test



Common errors in Unit test

(1) Misunderstood or incorrect arithmetic precedence,
(2) Mixed mode operations,
(3) Incorrect initialization,
(4) Precision inaccuracy,
(5) Incorrect symbolic representation of an expression.

Test cases should uncover errors such as

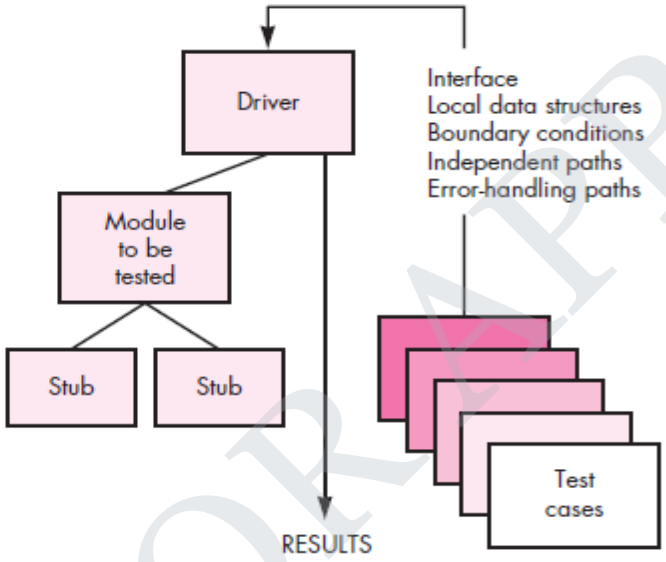
(1) Comparison of different data types,
(2) Incorrect logical operators or precedence,
(3) Incorrect comparison of variables,
(4) Improper or nonexistent loop termination,
(5) Improper loop termination

Unit Testing Procedures

A *driver* is a "main program" that accepts test case data, passes such data to the component and prints relevant results.

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.

FIGURE 17.4
Unit-test environment



INTEGRATION TESTING

"Big Bang" Approach:

All components are combined in advance. The entire program is tested as a whole.

Top-down integration:

Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. Referring to Figure 17.5

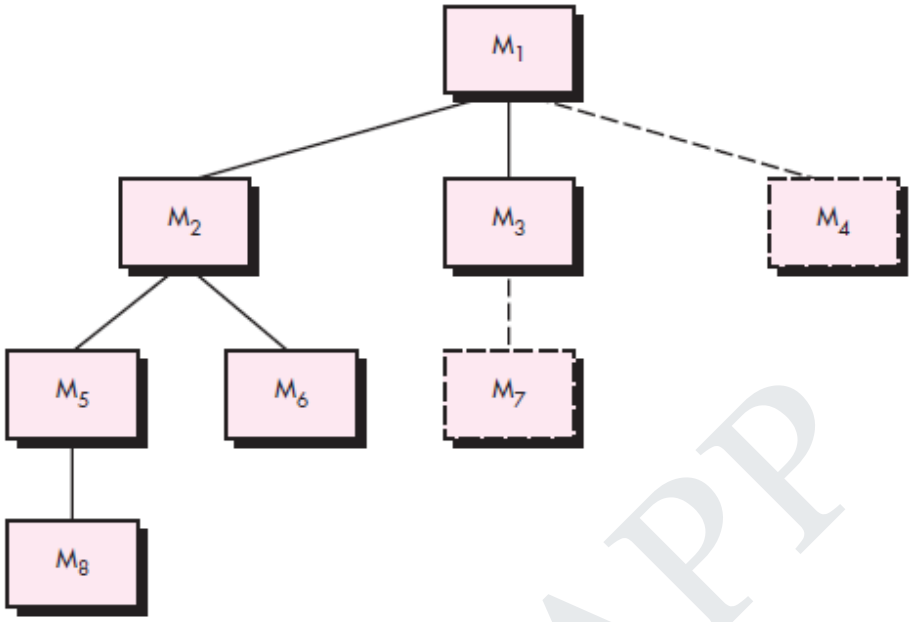
depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.

For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built.

Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

The major disadvantage of the top-down approach is the need for stubs and the attendant testing difficulties that can be associated with them.

FIGURE 17.5
Top-down integration



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

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

The process continues from step 2 until the entire program structure is built.

The most common of **these problems occurs** when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. As a tester, you are left with **three choices:**

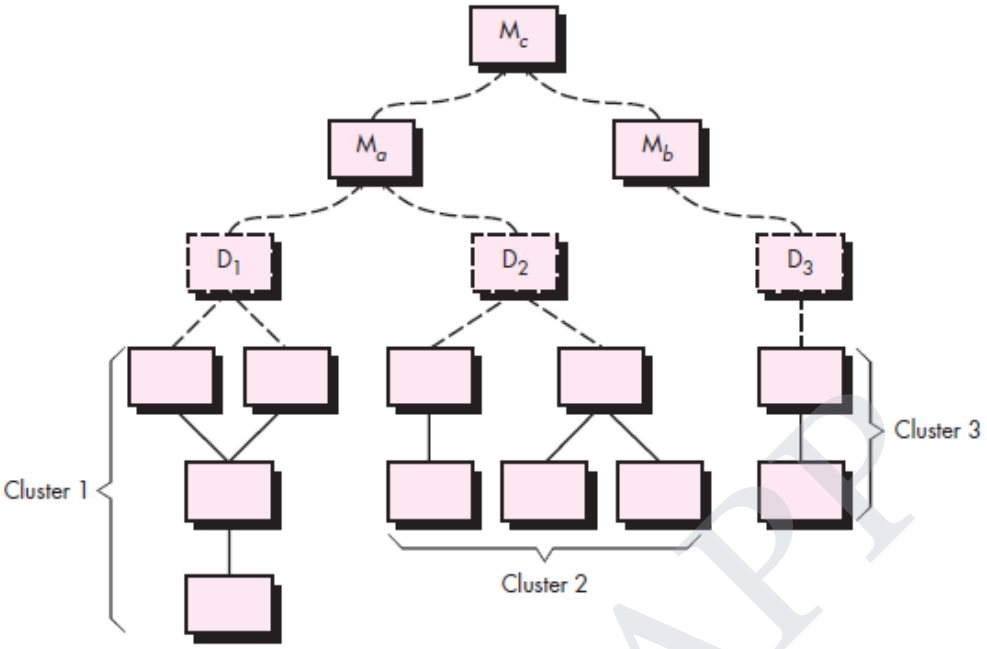
(1) delay many tests until stubs are replaced with actual modules,
(2) develop stubs that perform limited functions that simulate the actual module, or
(3) integrate the software from the bottom of the hierarchy upward.

Bottom-up integration:

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

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

FIGURE 17.6
Bottom-up integration



Integration follows the pattern illustrated in **Figure 17.6**. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M*a*. Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a*. Similarly, driver D3 for cluster 3 is removed prior to integration with module M*b*. Both M*a* and M*b* will ultimately be integrated with component M*c*, and so forth.

The major disadvantage of bottom-up integration is that "the program as an entity does not exist until the last module is added"

Sandwich Testing

In general, a combined approach (sometimes called *sandwich testing*) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

Regression testing.

In the context of an integration test strategy, *regression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

The *regression test suite* (the subset of tests to be executed) contains **three different classes of test cases:**

1. A representative sample of tests that will exercise all software functions.
2. Additional tests that focus on software functions that are likely to be affected by the change.
3. Tests that focus on the software components that have been changed.

Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools. *Capture/playback tools* enable the software engineer to capture test cases and results for subsequent playback and comparison.

Smoke testing:

Smoke testing is an integration testing approach that is commonly used when product software is developed. It allowing the software team to assess the project on a frequent basis.

In essence, the smoke-testing approach encompasses the **following activities:**

1. Software components that have been translated into code are integrated into **a *build.*** A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to **expose errors** that will keep the build from properly performing its function.
3. The build is integrated with other builds, and the entire product (in its current form) is **smoke tested daily**. The integration approach may be top down or bottom up.

Smoke testing provides a **number of benefits** when it is applied on complex, time critical software projects:

1. Integration risk is minimized.
2. The quality of the end product is improved.
3. Error diagnosis and correction are simplified.
4. Progress is easier to assess.

Test Specification

An overall plan for integration of the software and a description of specific tests is documented in a *Test Specification.*

A history of actual test results, problems, or peculiarities is recorded in a *Test Report* that can be appended to the *Test Specification,* if desired. Information contained in this section can be vital during software maintenance. Appropriate references and appendixes are also presented.

For example, integration testing for the *SafeHome* security system might be divided into the following test phases:

1. *User interaction* (command input and output, display representation, error processing and representation)
2. *Sensor processing* (acquisition of sensor output, determination of sensor conditions, actions required as a consequence of conditions)
3. *Communications functions* (ability to communicate with central monitoring station)
4. *Alarm processing* (tests of software actions that occur when an alarm is encountered)

The following criteria and corresponding tests are applied for all test phases:

1. *Interface integrity.* Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.
2. *Functional validity.* Tests designed to uncover functional errors are conducted.
3. *Information content.* Tests designed to uncover errors associated with local or global data structures are conducted.
4. *Performance.* Tests designed to verify performance bounds established during software design are conducted.

Validation	Verification
The process of checking that a system meets the needs and expectations of the customer.	The process of checking that a system meets its specification
Checks —Are we building the right product?	Checks —Are we building the product right?
Includes all the dynamic testing techniques.	Involves all the static testing techniques
Involves activities like Alpha testing, Beta testing and acceptance tesing	Involves activities like Unit, Integration and System testing.

VALIDATION TESTING

Software validation is achieved through a series of tests that demonstrate conformity**with requirements.**

Validation-Test Criteria:

A **test plan** outlines the classes of tests to be conducted, and a **test procedure** defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of **two possible** conditions exists:
(1) The function or performance characteristic conforms to specification and is accepted **or**
(2) a deviation from specification is uncovered and a deficiency list is created.

Configuration Review:

An important element of the validation process is a ***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 activities. The configuration review, sometimes called an **audit.**

Alpha Testing

The *alpha test* is conducted **at the developer's site** by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

Beta Testing

The *beta test* is conducted at one or more end-user sites. Unlike alpha testing, the **developer generally is not present**. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at **regular intervals**. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.

Acceptance testing

A variation on beta testing, called *customer acceptance testing,* is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors **before accepting the software from the developer**. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SYSTEM TESTING:

1)Recovery Testing:

Many computer-based systems must **recover from faults** and **resume processing** with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

If **recovery is automatic** (performed by the system itself), reinitialization, check pointing 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.

2)Security Testing

Security testing attempts to verify that **protection mechanisms** built into a system will, in fact, protect it from **improper penetration**. To quote Beizer "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."

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for **improper or illegal penetration**. **Penetration spans a broad range of activities:**

a. hackers who attempt to penetrate systems for sport,
b. disgruntled employees who attempt to penetrate for revenge,
c. dishonest individuals who attempt to penetrate for illicit personal gain.

3)Stress testing

It executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. **For example,**

(1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate

(2) input data rates may be increased by an order of magnitude to determine how input functions will respond

(3) test cases that require maximum memory or other resources are executed

(4) test cases that may cause thrashing in a virtual operating system are designed

(5) test cases that may cause excessive hunting for disk-resident data are created.

4)Performance testing

It is designed to test the **run-time performance** of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the **performance of an individual module** may be assessed as tests are conducted. However, it is not until all system elements are fully integrated that the true **performance of a system** can be ascertained.

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

5)Deployment Testing

Deployment testing, sometimes called ***configuration testing***, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

As an **example**, consider the Internet-accessible version of *SafeHome* software that would allow a customer to monitor the security system from remote locations. The *SafeHome* WebApp must be tested using all Web browsers that are likely to been countered. A more thorough deployment test might encompass combinations of Web browsers with various operating systems (e.g., Linux, Mac OS, Windows).Because security is a major issue, a complete set of security tests would be integrated with the deployment test.

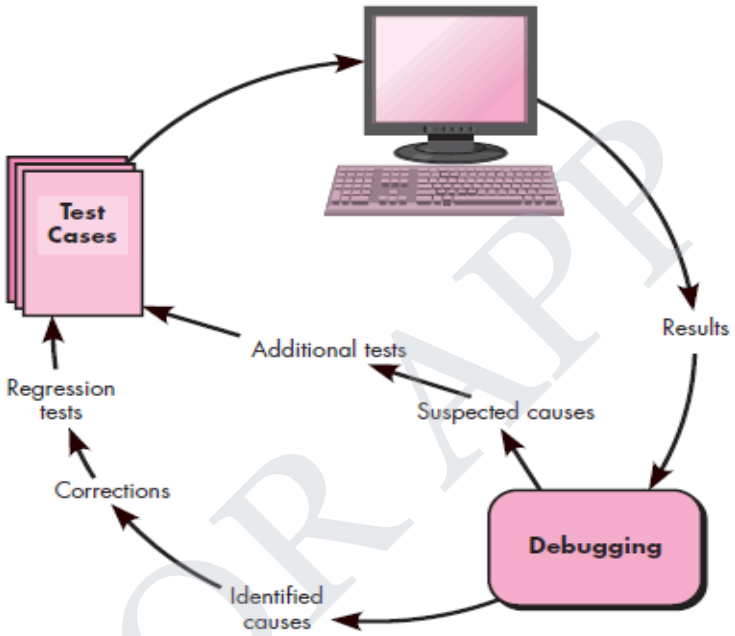
DEBUGGING

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

The Debugging Process

The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found.

FIGURE 17.7 The debugging process



Few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number oftasks running on different processors.

DEBUGGING STRATEGIES

Debugging tactics:

(1) brute force: The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error.

(2) backtracking,: *Back tracking* is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found.

(3) cause elimination: list of all possible causes is developed and tests are conducted to eliminate each.

Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Automated debugging:

Each of these debugging approaches can be supplemented with debugging tools that can provide you with semi automated support as debugging strategies are attempted. A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available.

The people factor:

Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! A fresh viewpoint, unclouded by hours of frustration, can do wonders.7 A final maxim for debugging might be: "When all else fails, get help!"

SOFTWARE IMPLEMENTATION TECHNIQUE

1. Definition phase - during this phase, plan the project, determine business objectives, and verify the feasibility of the project for given time, resource, and budget limits.
2. Operations Analysis phase - Includes documents business requirements, gaps in the software and system architecture requirements.
3. Solution Design phase - used to create designs for solutions that meet future business requirements and processes.
4. Build phase - during this phase coding and testing of customizations, enhancements, interfaces, and data conversions happens.
5. Transition phase - during this phase, the project team delivers the finished solution to the enterprise.
6. Production phase - Starts when the system goes live. Technical people work to stabilize and maintain the system under full transaction loads.

4.13.1. SELECTION OF IMPLEMENTATION METHODS

1. Factors affecting how much your project will cost and how much effort it will take
2. Techniques for project management and control
3. Planning project tasks and scope definitions
4. Analysis techniques for your business and technical systems
5. Designing ERP solutions for your business
6. Techniques to enable the system
7. Managing change, transitioning from old to new systems, and supporting users after your systems "go live"

4.13.2. RAPID IMPLEMENTATION TECHNIQUES

Rapid implementations focus on delivering a predefined set of functionality. A key set of business processes is installed in a standard way to accelerate the implementation schedule.

CODING PRACTICES

Best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software.

Good programming practices are techniques that you can follow to create the best code. Programming practices cover everything from making code more readable to creating code with faster performance.

Coding Principles.

The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

Preparation principles: ***Before you write one line of code, be sure you***

- Understand of the problem you're trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

Programming principles: ***As you begin writing code, be sure you***

- Constrain your algorithms by following structured programming practice.
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards.
- Write code that is self-documenting.
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Validation Principles: ***After you've completed your first coding pass, be sure you***

- Conduct a code walkthrough when appropriate.
- Perform unit tests and correct errors you've uncovered.
- Refactor the code.

REFACTORING

Refactoring allows a software engineer to improve the internal structure of a design (or source code) without changing its external functionality or behavior.

In essence, refactoring can be used to improve the efficiency, readability, or performance of a design or the code that implements a design.

4.15.1. REFACTORING TECHNIQUES

Broadly refactoring can be divided in the following categories:

- **Project/Program Structural refactoring**: It includes code refactoring to achieve better program structure. Movement of methods and classes to more logical units.
- **Code Clean Up Refactoring:** It includes code refactoring to achieve removal of unused code and classes, renaming of classes, methods and variables which are misleading or confusing.
- **Code Standard Refactoring:** It includes code refactoring to achieve the quality code. Examples are use of map keyset iterator instead of using entry-set iterator to get the key/value pair in the code. There are automated review tools which can be integrated with the IDE e.g. PMD and provides the potential defects. Developers should regularly refactor the code as per the standard code lines.
- **User Interface Refactoring:** Changing the UI technology without affecting the functionality incrementally.
- **Database Clean up Refactoring:** It includes cleaning of unnecessary and redundant data without changing the data architecture. This includes data migration as well as data cleaning.
- **Database Design & Schema Refactoring:** This task includes enhancing the database schema leaving the actual fields required by the application intact.
- **Architecture Refactoring:** It includes modularization of application. Architecture refactoring is achieved by code slicing, application re-aggregation and consolidation. Architecture driven refactoring is targeted to achieve certain business objectives where existing practices fails to deliver those objectives. Substitute Algorithm, Split Loop, Introduce Assertion, the list goes on & on…

4.15.2. WHY REFACTORING NEEDED?

Software refactoring or rewriting becomes essential for the organization when following problems becomes visible in the software:

- Maintainability - Code is not easily maintainable
- Extendibility - Extending/adding new features in the application are not possible or very expensive.
- It can be due to various reasons.

Example

- **Lack of Modularity** - Existing feature of one application can't be used in another application due to its tightly coupling with the application components
- **Lack of Reusable Components** – There are instances of code duplicity and potential reusable components dependency on application code.
- **Lack of Pluggable Components** - Existing components are not easily replaceable due to its application codes tightly coupling with the component
- **Service Oriented Architecture** – Scope for SOA components where each component can work as a service and reusable
- **Code redundancy** – Application has lots of dead code and duplicate code
- **Lack of Layered Architecture** - Any change in one layer causing changes in all other layers
- **Poor Coding Style** - Coding standards has not been followed properly – It includes improper names to object/methods, accessing the fields without getter/setters
- **Illogical Methods Composition** – Illogical grouping of methods in one class..
- **Improper Packaging** - Artifacts are placed in the application code which can be kept at other locations; forcing developer to change the jars in each of the application manually instead of updating it a centralized location.
- **Use of Old Version of third party application/jars** – Application is using older version of software's instead of using latest version and hence new features can't be used and explored in the application.

UNIT V
PROJECT MANAGEMENT

Software Project Management: Estimation – LOC, FP Based Estimation, Make/Buy Decision COCOMO I & II Model – Project Scheduling – Scheduling, Earned Value Analysis Planning – Project Plan, Planning Process, RFP Risk Management – Identification, Projection - Risk Management-Risk Identification-RMMM Plan-CASE TOOLS.

ESTIMATION

- S/W is the most expensive element of virtually all computer based systems
- The accuracy of a s/w project estimate is predicated on a number of things:
  - ✓ The degree to which the planner has properly estimated the size of the product to be built
  - ✓ The ability to translate the size estimate into human effort, calendar time, and dollars (required availability of past records)
  - ✓ The degree to which the project plan reflects the abilities of the s/w team
  - ✓ The stability of product requirements and the environment that supports the s/w engineering effort
- Sizing represents the project planner's first major challenge
- Size refers to a quantifiable outcome of the s/w project (e.g. LOC and/or FP)

Size-Oriented Metrics

- LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists.
- The planner must estimate the LOC to be produced long before analysis and design has been completed.

Function-Oriented Metrics

- Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity.
- The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.
- Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

Reconciling LOC and FP Metrics

The relationship between **lines of code** and **function points** depends upon the programming language that is used to implement the software and the quality of the design. Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation, an historical baseline of information must be established.

Within the context of process and project metrics, productivity and quality should be concerned which are the measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For process improvement and project planning purposes, the interest is historical.

5.5.3 LOC based Estimation

- Size oriented measure is derived by considering the size of software that has been produced.
- The organization builds a simple record of size measure for the software projects. It is built on past experiences of organizations.
- It is a direct measure of software
- Using historical data the project planner **expected value** by considering following variables -
  1. Optimistic
  2. Most likely
  3. Pessimistic

For example, following formula

$$S = [S_{opt} + 4 * S_m + S_{pess}]/6$$

considers for "most likely" estimate where S is the estimation size variable, S_{opt} represents the optimistic estimate, S_m represents the most likely estimate and S_{pess} represents the pessimistic estimate values.

- A simple set of size measure that can be developed is as given below :
  - Size = Kilo Lines of Code (KLOC)
  - Effort = Person/month
  - Productivity = KLOC/person-month
  - Quality = Number of faults/KLOC
  - Cost = $/KLOC
  - Documentation = Pages of documentation/KLOC
- The size measure is based on the lines of code computation. The lines of code is defined as one line of text in a source file.
- While counting the lines of code the simplest standard is :
  - Don't count blank lines.
  - Don't count comments.
  - Count everything else.
- The size oriented measure is not universally accepted method.

Advantages

1. Artifact of software development which is easily counted.
2. Many existing methods use LOC as a key input.
3. A large body of literature and data based on LOC already exists.

Disadvantages

1. This measure is dependent upon the programming language.
2. This method is well designed but shorter program may get suffered.
3. It does not accommodate non procedural languages.
4. In early stage of development it is difficult to estimate LOC.

Example

Consider an ABC project with some important modules such as

1. User interface and control facilities
2. 2D graphics analysis
3. 3D graphics analysis
4. Database management
5. Computer graphics display facility
6. Peripheral control function
7. Design analysis models

Estimate the project in based on LOC

Solution

For estimating the given application we consider each module as separate function and corresponding lines of code can be estimated in the following table as

Function	Estimated LOC
User Interface and Control Facilities(UICF)	2500
2D graphics analysis(2DGA)	5600
3D Geometric Analysis function(3DGA)	6450
Database Management(DBM)	3100
Computer Graphics Display Facility(CGDF)	4740
Peripheral Control Function(PCF)	2250
Design Analysis Modules (DAM)	7980
Total estimation in LOC	**32620**

- Expected LOC for 3D Geometric analysis function based on three point estimation is -
  - Optimistic estimation 4700
  - Most likely estimation 6000
  - Pessimistic estimation 10000

$$S = [S_{opt} + (4 * S_m) + S_{pess}]/6$$

Expected value = [4700 + (4 * 6000) + 10000] /6 → 6450

- A review of historical data indicates -
  1. Average productivity is 500 LOC per month
  2. Average labor cost is $6000 per month

Then cost for lines of code can be estimated as

cost/LOC = (6000/500) = $12

By considering total estimated LOC as 32620

- Total estimated project cost = (32620*12) = $391440
- Total estimated project effort = (32620/500) = 65 Person-months

Example:2

Consider a real-time example scenario:

The mechanical CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

This statement of scope is preliminary—it is not bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin, the planner must determine what "characteristics of good human/machine interface design" means or what the size and sophistication of the "CAD database" are to be. For our purposes, assume that further refinement has occurred and that the major software functions are identified. Following the decomposition technique for LOC, an estimation table is developed.

A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic, 4600 LOC; most likely, 6900 LOC; and pessimistic, 8600 LOC. Applying equation, the expected value for the 3D geometric analysis function is 6800 LOC.

FIGURE 26.2
Estimation table for the LOC methods

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

Other estimates are derived in a similar fashion. By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system.

A review of historical data indicates that the organizational **average productivity** for systems of this type is **620 LOC/pm.**

Based on a burdened **labour rate** of **$8000 per month**, **the cost per line of code is approximately $13.**

Based on the LOC estimate and the historical productivity data, the total estimated **project cost is $431,000 and the estimated effort is 54 person-months.**

5.5.5 Function Oriented Metrics

- The function point model is based on functionality of the delivered application.
- These are generally independent of the programming language used.
- This method is developed by Albrecht in 1979 for IBM.
- Function points are derived using :
  1. Countable measures of the software requirements domain
  2. Assessments of the software complexity.

How to calculate function point ?

- The data for following information domain characteristics are collected :

1. Number of user inputs - Each user input which provides distinct application data to the software is counted.
2. Number of user outputs - Each user output that provides application data to the user is counted, e.g. screens, reports, error messages.
3. Number of user inquiries - An on-line input that results in the generation of some immediate software response in the form of an output.
4. Number of files - Each logical master file, i.e. a logical grouping of data that may be part of a database or a separate file.

5. Number of external interfaces - All machine-readable interfaces that are used to transmit information to another system are counted.

- The organization needs to develop criteria which determine whether a particular entry is simple, average or complex.
- The weighting factors should be determined by observations or by experiments.

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total	→						

Example for FP Based Estimation

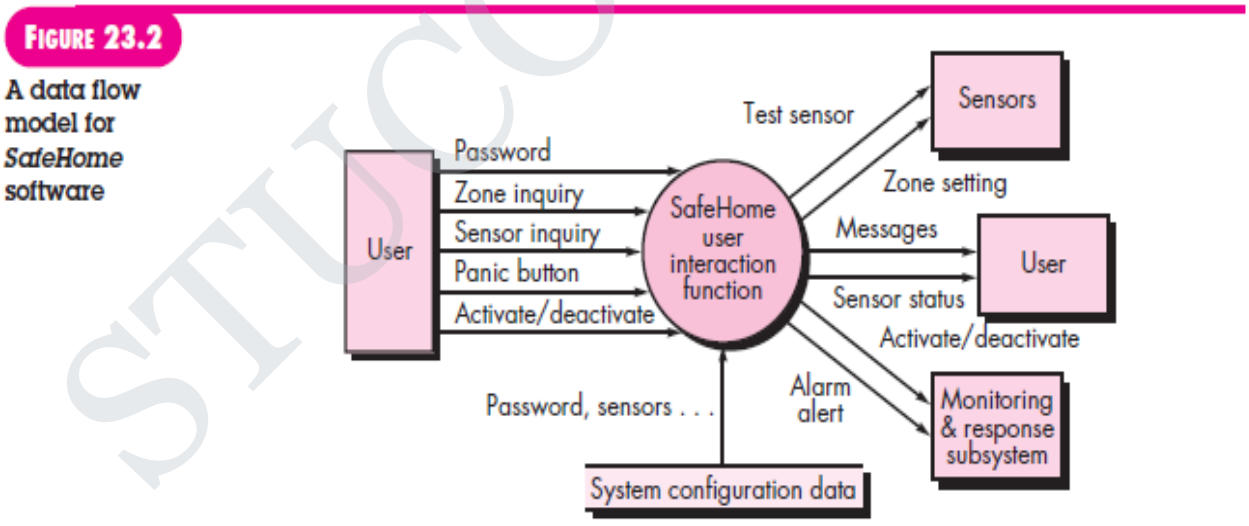


FIGURE 23.2 A data flow model for *SafeHome* software

Three external inputs—password, panic button, and activate/deactivate
Two external inquiries—zone inquiry and sensor inquiry
One ILF -system configuration file
Two external outputs - messages and sensor status
Four EIFs - test sensor, zone setting, activate/deactivate, and alarm alert

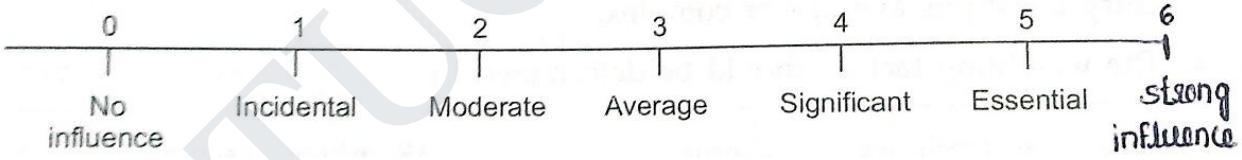
FIGURE 23.3 Computing function points

Information Domain Value	Count		Weighting factor: Simple	Average	Complex		
External Inputs (EIs)	3	×	3	4	6	=	9
External Outputs (EOs)	2	×	4	5	7	=	8
External Inquiries (EQs)	2	×	3	4	6	=	6
Internal Logical Files (ILFs)	1	×	7	10	15	=	7
External Interface Files (EIFs)	4	×	5	7	10	=	20
Count total							50

we assume that $\Sigma(F_i)$ is 46

$$FP = 50 \times [0.65 + (0.01 \times 46)] = 56$$

- Once the functional point is calculated then we can compute various measures as follows
  - Productivity = FP/person-month
  - Quality = Number of faults/FP
  - Cost = $/FP
  - Documentation = Pages of documentation/FP.



Advantages

1. This method is independent of programming languages.
2. It is based on the data which can be obtained in early stage of project .

Disadvantages

1) This method is more suitable for business systems and can be developed for that domain.

2) Many aspects of this method are not validated.

3) The functional point has no significant meaning. It is just a numerical value.

5.5.6 Example of FP based Estimation

FP focuses on information domain values rather than software functions. Thus we create a function point calculation table for ABC project.

INFO DOMAIN VALUE	Opt.	most likely	pessimistic		esti. value		weight factor		FP
NO.OF INPUTS	25	28	32	⇒	28.1	×	4	=	112
NO. OF OUTPUTS	14	17	20	⇒	17	×	5	=	85
NO. OF INQURIES	17	23	30	⇒	23.1	×	5	=	116
NO. OF FILES	5	5	7	⇒	5.33	×	10	=	53
NO. OF EXTERNAL INTERFACES	2	2	3	⇒	2	×	7	=	15
COUNT TOTAL									381

- For this example we assume average complexity weighting factor.
- Each of the complexity weighting factor is estimated and the complexity adjustment factor is computed using the complexity factor table below. (Based on the 14 questions)

Sr. No.	FACTOR	VALUE (Fi) → range(0-6)
1.	Back-up and recovery ?	4
2.	Data communication ?	2
3.	Distributed processing ?	0
4.	Performance critical ?	4
5.	Existing operational environment ?	3
6.	On-line data entry ?	4
7.	Input transactions over multiple screens?	5
8.	Online updates ?	3
9.	Information domain values complex ?	5
10	Internal processing complex?	5
11.	Code designed for reuse?	4
12.	Conversion / installation in design?	3
13.	Multiple installations?	5
14.	Application designed for change ?	5 → range(0-84)

$\sum$ (Fi) → 52

The estimated number of adjusted FP is derived using the following formula : -

FP ESTIMATED = (FP COUNT TOTAL * [COMPLEXITY ADJUSTMENT FACTOR]

FP ESTIMATED = COUNT TOTAL* [0.65 + (0.01* $\sum$ (Fi))]

Complexity adjustment factor = [0.65 + (0.01 * 52)] = 1.17

- FP ESTIMATED = (381 * 1.17) = 446 (Function point count adjusted with complexity adjustment factor)
- A review of historical data indicates -
  1. Average productivity is 6.5 FP/Person month
  2. Average labor cost is $6000 per month
- Calculations for cost per function point, total estimated project cost and total effort
  1. The cost per function point = (6000 / 6.5) = $923
  2. Total estimated project cost = (446 * 923) = $411658
  3. Total estimated effort = (446 / 6.5) = 69 Person-month.

MAKE/BUY DECISION:

Software engineering managers are faced with a make/ buy decision that can be further complicated by a number of acquisition options:

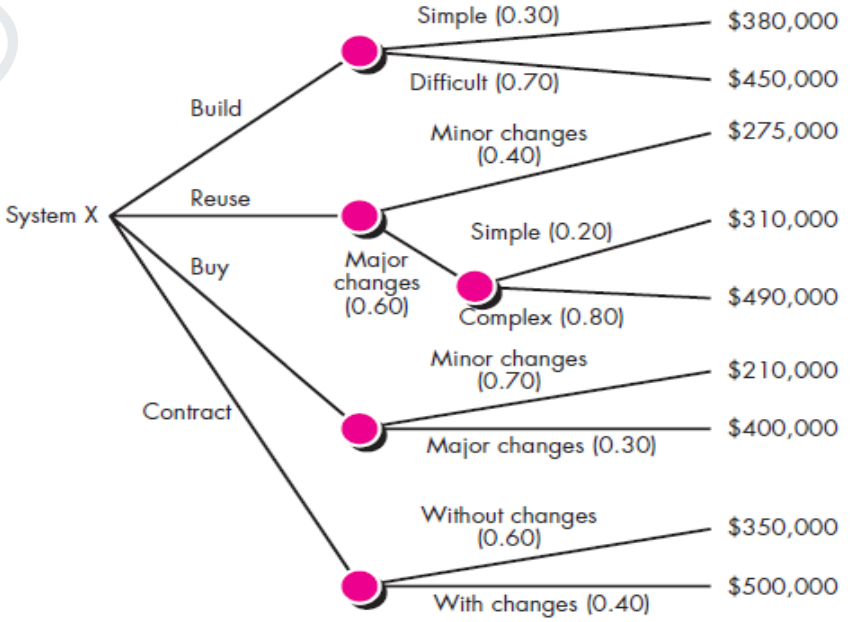
(1) software may be purchased (or licensed) off-the-shelf,

(2) "full-experience" or "partial-experience" software components may be acquired and then modified and integrated to meet specific needs, or

(3) software may be custom built by an outside contractor to meet the purchaser's specifications.

Creating a Decision Tree

FIGURE 26.8
A decision tree to support the make/buy decision



decision tree for a software based system X. In this case, the software engineering organization can (1) build system X from scratch, (2) reuse existing partial-experience components to construct the system, (3) buy an available software product and modify it to meet local needs, or (4) contract the software development to an outside vendor.

If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult. project planner estimates that a difficult development effort will cost $450,000.A "simple" development effort is estimated to cost $380,000. The expected value for cost, computed along any branch of the decision tree, is

$$\text{Expected cost} = \Sigma\,(\text{path probability})_i \times (\text{estimated path cost})_i$$

where *i* is the decision tree path. For the build path,

$$\text{Expected cost}_{\text{build}} = 0.30\ (\$380K) + 0.70\ (\$450K) = \$429K$$

$$\text{Expected cost}_{\text{reuse}} = 0.40\ (\$275K) + 0.60\ [0.20\ (\$310K) + 0.80\ (\$490K)] = \$382K$$
$$\text{Expected cost}_{\text{buy}} = 0.70\ (\$210K) + 0.30\ (\$400K) = \$267K$$
$$\text{Expected cost}_{\text{contract}} = 0.60\ (\$350K) + 0.40\ (\$500K) = \$410K$$

Based on the probability and projected costs that have been noted in Figure 26.8, the lowest expected cost is the "buy" option. It is important to note, however, that many criteria—not just cost— must be considered during the decision-making process. Availability, experience of the developer/ vendor/contractor, conformance to requirements, local "politics," and the likelihood of change are but a few of the criteria that may affect the ultimate decision to build, reuse, buy, or contract.

COst COnstructive Model (COCOMO)

The Constructive Cost Model (COCOMO) is an empirical estimation model i.e., the model uses a theoretically derived formula to predict cost related factors. This model was created by **"Barry Boehm".**

The COCOMO model consists of **three models**:-

1. **The Basic COCOMO model:** - It computes the effort & related cost applied on the software development process as a function of program size expressed in terms of estimated lines of code (LOC or KLOC).
2. **The Intermediate COCMO model:** - It computes the software development effort as a function of – a) Program size, and b) A set of cost drivers that includes subjective assessments of product, hardware, personnel, and project attributes.
3. **The Detailed COCOMO model:** - It incorporates all the characteristics of the intermediate version along with an assessment of cost driver's impact on each step of software engineering process.

COCOMO applies to **three classes of software projects**:

1. **Organic projects** - "small" teams with "good" experience working with "less than rigid" requirements
2. **Semi-detached projects** - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
3. **Embedded projects** - developed within a set of "tight" constraints. It is also combination of organic and semi-detached projects.(hardware, software, operational, ...)

Basic COCOMO

Basic COCOMO computes software development effort (and cost) as a function of program size. Program size is expressed in estimated thousands of source lines of code (SLOC, KLOC).

The basic COCOMO equations take the form

Effort Applied (E) = $a_b(KLOC)^{b}_{b}$ **[person-months]**

Development Time (D) = $c_b(\text{Effort Applied})^{d}_{b}$ **[months]**

Staff size(SS)= Effort Applied / Development Time **[Person]**

Productivity(P) = KLOC/E

where, **KLOC** is the estimated number of delivered lines (expressed in thousands) of code for project. The coefficients a_b, b_b, c_b and d_b are given in the following table:

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

Merits of basic COCOMO model:

1. Basic COCOMO is good for quick estimate of software costs.

Limitation of basic model:

1. The accuracy of this model is limited because it does not consider certain factors for cost estimation of software. These factors are hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

Intermediate COCOMO

The Intermediate **COCOMO** formula now takes the form:

$E=a_i$ (KLOC) ^{bi} (EAF)

Development Time (D) = $c_b(\text{Effort Applied})^{d}_{b}$ **[months]**

Staff size(SS)= Effort Applied / Development Time **[Person]**

Productivity(P) = KLOC/E

where E is the effort applied in person-months, **KLOC** is the estimated number of thousands of delivered lines of code for the project, and **EAF** is the factor calculated above. The coefficient $\mathbf{a_i}$ and the exponent $\mathbf{b_i}$ are given in the next table.

Software project	a_i	b_i	c_i	d_i
Organic	**3.2**	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	**2.8**	1.20	2.5	0.32

The Development time **D** calculation uses **E** in the same way as in the Basic COCOMO. Staff Size (**SS**) also computed as in the basic model

Intermediate *COCOMO* computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a **set of four "cost drivers"**, each with a number of subsidiary attributes:-

Each of the **15 attributes** receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value). An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an ***effort adjustment factor (EAF).*** Typical values for EAF range from 0.9 to 1.4.

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
1. Required software reliability	0.75	0.88	1.00	1.15	1.40	
2. Size of application database		0.94	1.00	1.08	1.16	
3. Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
4. Run-time performance constraints			1.00	1.11	1.30	1.66
5. Memory constraints			1.00	1.06	1.21	1.56
6. Volatility of virtual machine environment		0.87	1.00	1.15	1.30	
7. Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
8. Analyst capability	1.46	1.19	1.00	0.86	0.71	
9. Applications experience	1.29	1.13	1.00	0.91	0.82	
10. Software engineer capability	1.42	1.17	1.00	0.86	0.70	
11. Virtual machine experience	1.21	1.10	1.00	0.90		
12. Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
13. Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
14. Use of software tools	1.24	1.10	1.00	0.91	0.83	
15. Required development schedule	1.23	1.08	1.00	1.04	1.10	

Merits of Intermediate model:

1. This model can be applied to almost entire software product for easy and rough cost estimation during early stage
2. It can be applied at the software product component level for obtaining more accurate cost estimation.

Limitation of Intermediate model:

1. A product with many components is difficult to estimate.
2. The effort multipliers are not dependent on phases
3. The estimation is within 20% of actual 68% of the time

Detailed COCOMO

- Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.
- The detailed model uses different effort multipliers for each cost driver attribute. These **Phase Sensitive** effort multipliers are each to determine the amount of effort required to complete each phase.
- In detailed cocomo, the whole software is divided in different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort
- In detailed COCOMO, the effort is calculated as function of program size and a set of cost drivers given according to each phase of software life cycle.
- A Detailed project schedule is never static.
- The phases of detailed COCOMO are:-
  - ✓ plan and requirement.
  - ✓ system design.
  - ✓ detailed design.
  - ✓ module code and test.
  - ✓ integration and test.

COCOMO II MODEL

Barry Boehm introduced a hierarchy of software estimation models bearing the name COCOMO, for ***COnstructive COst MOdel***. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMOII

COCOMO II is actually a hierarchy of estimation models that address the following areas:

- ***Application composition model***. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
- ***Early design stage model.*** Used once requirements have been stabilized and basic software architecture has been established.
- ***Post-architecture-stage model.*** Used during the construction of the software.

COCOMO II models require sizing information. **Three different sizing options** are available as part of the model hierarchy:

1. object points
2. function points
3. lines of source code.

The COCOMO II application composition model **uses object points**

FIGURE 26.6 Complexity weighting for object types. Source: [Boe96].

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

the *object point* is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e.,simple, medium, or difficult)

complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

$$\text{NOP} = (\text{object points}) \times [(100 - \%\text{reuse})/100]$$

where NOP is defined as new object points. To derive an estimate of effort based on the computed NOP value, a "productivity rate" must be derived. Figure 26.7 presents the productivity rate

FIGURE 26.7 Productivity rate for object points. Source: [Boe96].

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

$$\text{PROD} = \frac{\text{NOP}}{\text{person-month}}$$

for different levels of developer experience and development environment maturity. Once the productivity rate has been determined, an estimate of project effort is computed using

$$\text{Estimated effort} = \frac{\text{NOP}}{\text{PROD}}$$

EARNED VALUE ANALYSIS

The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total.

To determine the earned value, the following steps are performed:

1. The **budgeted cost of work scheduled (BCWS or PV)** is determined for each work task represented in the schedule.
2. The BCWS values for all work tasks are summed to derive the **budget at completion (BAC)**. Hence,

   $$\mathbf{BAC} = \sum (BCWS_k) \text{ for all task } k$$

3. Next, the value for **budgeted cost of work performed (BCWP or EV)** is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

   Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

$$\text{Schedule performance index, SPI} = \frac{BCWP}{BCWS}$$

$$\text{Schedule Variance, SV} = BCWP - BCWS$$

$$\text{Percent scheduled for completion} = \frac{BCWS}{BAC}$$

$$\text{Percent complete} = \frac{BCWP}{BAC}$$

$$\text{Cost performance index, CPI} = \frac{BCWP}{ACWP}$$

$$\text{Cost variance, CV} = BCWP - ACWP$$

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project. Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.

EARNED VALUE ANALYSIS

PROBLEM:

Given the following project plan of tables table1 and table2: Table1

ID	Task	Predecessor(*)	Expected duration(days)	Budget($)
A	Meet with client		5	500
B	Write SW	A	20	10000
C	Debug SW	B	5	1500
D	Prepare draft manual	B	5	1000
E	Meet with clients	D	5	1000
F	Test SW	C,E	20	2000
G	Make modifications	F	10	8000
H	Finalize manual	G	10	5000
I	Advertise	C,E	20	8000

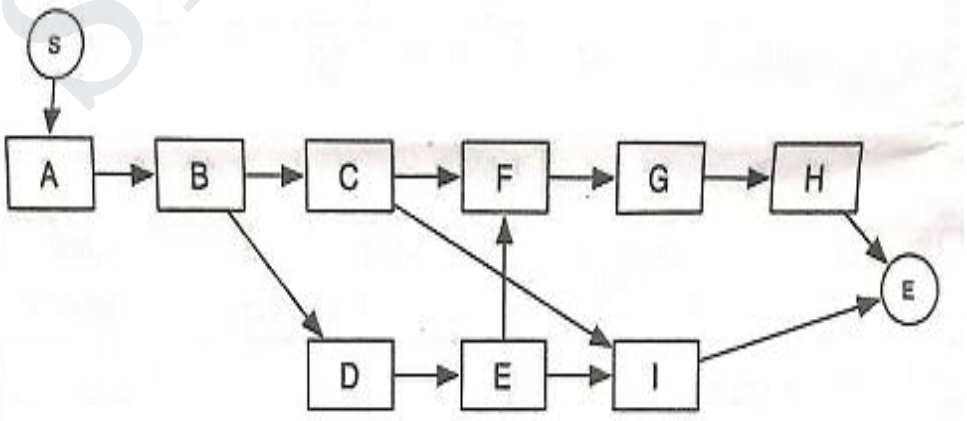
(*)all dependencies are assumed to be FS-Finish to Start And the following progress status: And the following progress status: **Table2**

ID	Task	Status	Actual start(days)	Actual duration(days)	Actual Cost($)
A	Meet with client	100%			1500
B	Write SW	100%	+ 5days	+10 days	9000
C	Debug SW	100%	+ 15days	+5 days	2500
D	Prepare draft manual	100%	As per other delays		1000
E	Meet with clients	100%	As per other delays		1000
F	Test SW	100%	As per other delays		750
G	Make modifications	0%	As per other delays		0
H	Finalize manual	0%	As per other delays		0
I	Advertise	10%	+15 on top of other delays		1000

Perform an analysis of the project status at week 13,using EVA .Use the CPI and SPI to determine project efficiency. Explain the process involved . (APR/MAY 2017)

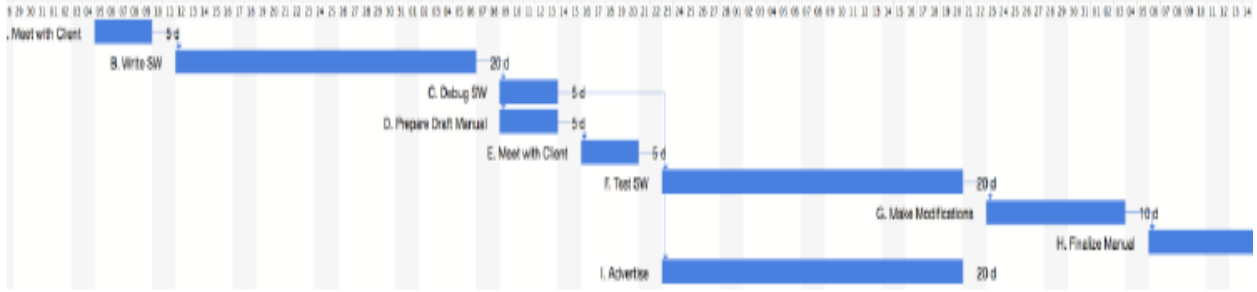
Solution:

1. Construct the task network

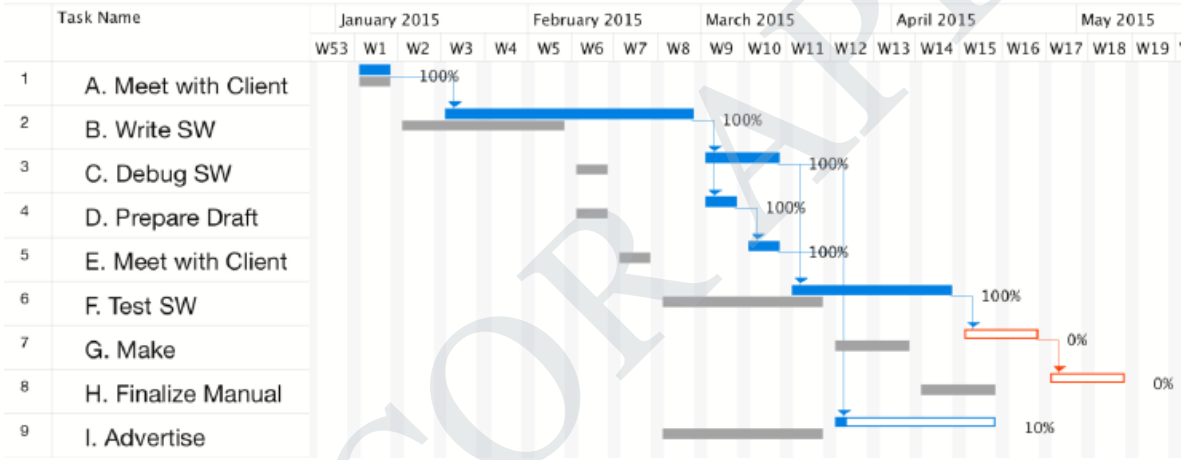


b

2. Drawing the Gantt chart of the plan



3. Drawing the Gantt chart of the actual plan (progress status)



4. **Perform the analysis (plot PV, AC, EV, CPI, SPI)**

i. **PV** is the sum of planned costs. (ie, summing and cumulating them **over time (ie,w13)**

		W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13
A	Meet with client	500												
B	Write SW		2500	2500	2500	2500								
C	Debug SW						1500							
D	Prepare draft manual						1000							
E	Meet with clients							1000						
F	Test SW								500	500	500	500		
G	Make modifications												4000	4000
H	Finalize manual													
I	Advertise								2000	2000	2000	2000		
	Total	500	2500	2500	2500	2500	2500	1000	2500	2500	2500	2500	4000	4000
	Planned Value	500	3000	5500	8000	10500	13000	14000	16500	19000	21500	24000	28000	32000

ii. **AC is the sum of the actual costs**(ie, summing and cumulating them **over time (ie,w13)**

For each activity, we look at its actual costs (second table of the question) and split them evenly for the actual duration of the activity, up to the monitoring date (that is, the date in which the analysis is performed)

	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	Total
A Meet with client	1500													1500
B Write SW			1500	1500	1500	1500	1500	1500						9000
C Debug SW									1250	1250				2500
D Prepare draft manual									1000					1000
E Meet with clients										1000				1000
F Test SW											250	250	250	750
G Make modifications														0
H Finalize manual														0
I Advertise												500	500	1000
Total	1500	0	1500	1500	1500	1500	1500	1500	2250	2250	250	750	750	
AC	1500	1500	3000	4500	6000	7500	9000	10500	12750	15000	15250	16000	16750	

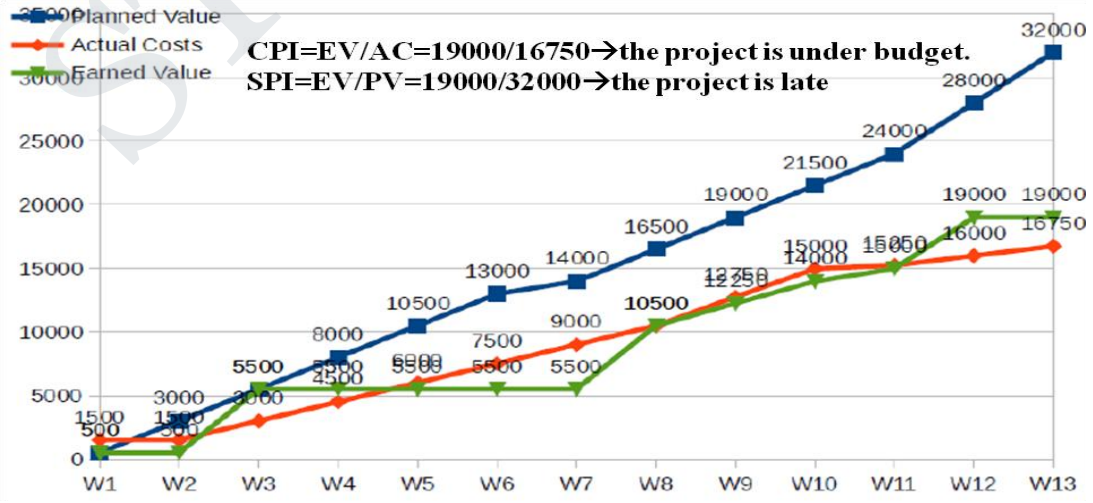
iii. **EV is the sum of the planned costs on the actual schedule.** (ie, summing and cumulating them **over time(ie,w13)**

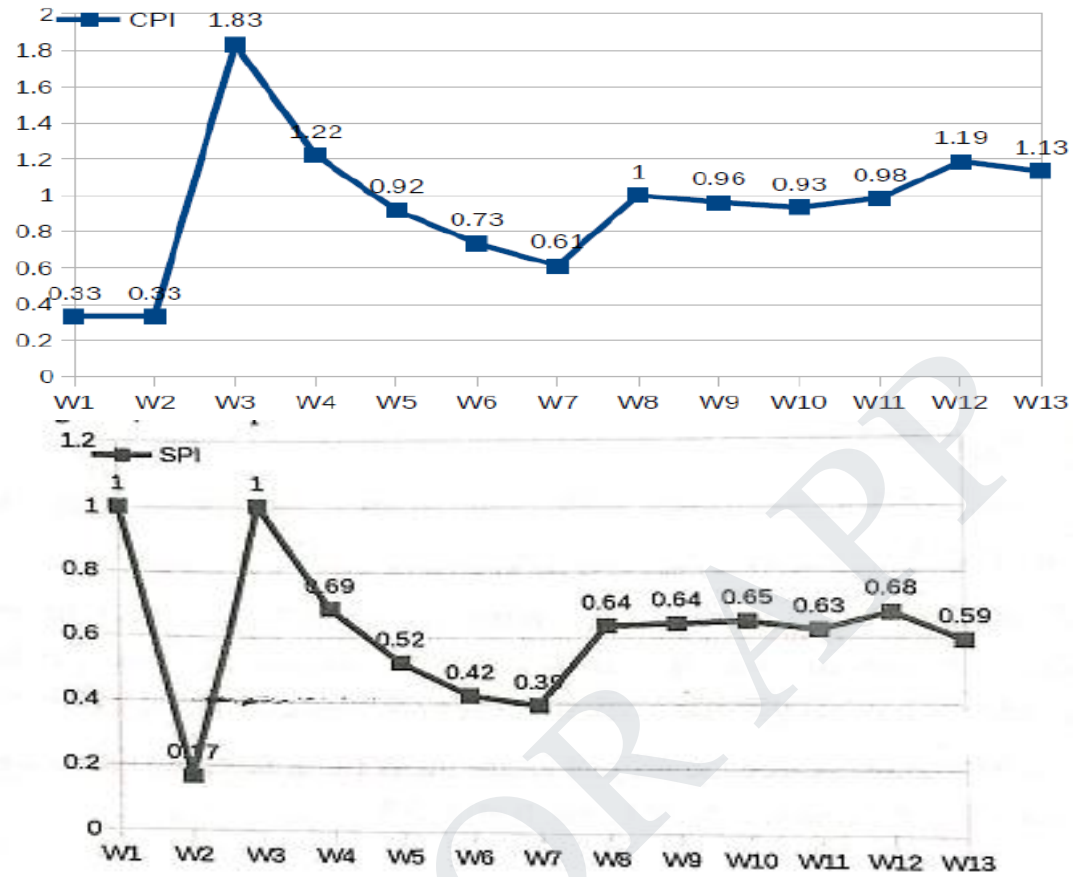
There are different rules for computing EV. We use 50%-50% (50% of planned costs when an activity starts, the remaining 50%,when the activity ends.

	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	Total
A Meet with client	500													500
B Write SW			5000	0	0	0	0	5000						10000
C Debug SW									750	750				1500
D Prepare draft manual									1000					1000
E Meet with clients										1000				1000
F Test SW											1000	0	0	1000
G Make modifications														0
H Finalize manual														0
I Advertise												4000	0	4000
Total	500	0	5000	0	0	0	0	5000	1750	1750	1000	4000	0	
Earned Value	500	500	5500	5500	5500	5500	5500	10500	12250	14000	15000	19000	19000	

iv) CPI & SPI Analysis:

From the data at W13 we can observe the following:

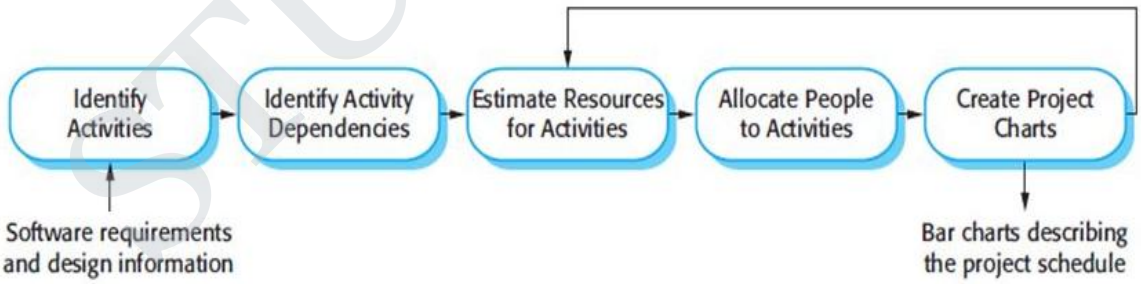




PROJECT SCHEDULING

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.

Project scheduling process:



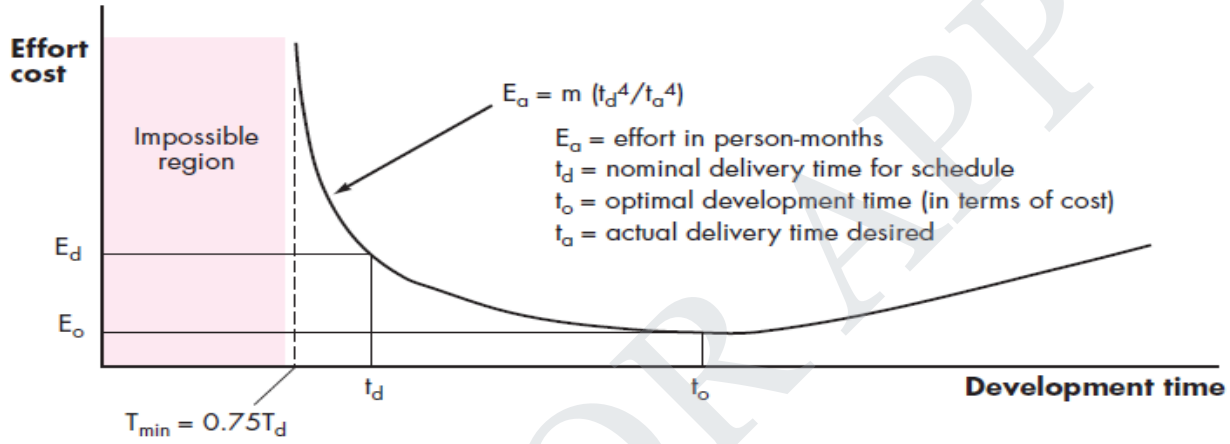
Basic Principles

- **Compartmentalization.** The project must be compartmentalized into a number of manageable activities and tasks
- **Interdependency.** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel.
- **Time allocation.** Each task must be assigned a start date and a completion date.
- **Effort validation.** Every project has a defined number of people on the software team.

- **Defined responsibilities.** Every task that is scheduled should be assigned to a specific team member.
- **Defined outcomes.** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product.
- **Defined milestones.** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality.

The Relationship between People and Effort

In small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.



The **Putnam-Norden-Rayleigh (PNR) Curve** provides an indication of the relationship between effort applied and delivery time for a software project. The curve indicates a **minimum value** to that indicates the **least cost for delivery**.

$$t_o = 2t_d.$$

The number of delivered lines of code (source statements), L, is related to effort and development time by the equation:

$$L = P \times E^{1/3} t^{4/3}$$

where E is development effort in person-months, P is a productivity parameter that reflects a variety of factors that lead to high-quality software engineering work (typical values for P range between 2000 and 12,000), and t is the project duration in calendar months.

Rearranging this software equation, we can arrive at an expression for development effort *E*:

$$E = \frac{L^3}{P^3 t^4}$$

Effort Distribution

A recommended distribution of effort across the software process is often referred to as the **40–20–40 rule.** Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. 20 percent of effort is deemphasized in the coding.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

DEFINING A TASK SET FOR THE SOFTWARE PROJECT

Task Set

- An effective software process should define a collection of task sets.
- A task set is a collection of software engineering work tasks, milestones, and deliverables.
- Tasks sets are designed to accommodate different types of projects.
- Typical **project types:**
  1. **Concept development projects**
  2. **New application development projects**
  3. **Application enhancement projects**
  4. **Application maintenance projects**
  5. **Reengineering projects**

Task Set example:

Concept development projects are approached by applying the following **actions:**

Actions 1.1 Concept scoping determines the overall scope of the project.

Actions 1.2 Preliminary concept planning establishes the organization's ability to undertake the work implied by the project scope.

Actions 1.3 Technology risk assessment evaluates the risk associated with the Technology.

Actions 1.4 Proof of concept demonstrates the viability of a new technology

Actions 1.5 Concept implementation implements the concept.

Actions 1.6 Customer reaction to the concept solicits feedback on a new technology concept

Refinement of Software Engineering Actions

Refinement begins by taking each action and decomposing it into a set of tasks (with related work products and milestones). The software engineering actions described in the preceding section may be used to define a macroscopic schedule for a project. However, the macroscopic schedule must be refined to create a detailed project schedule.

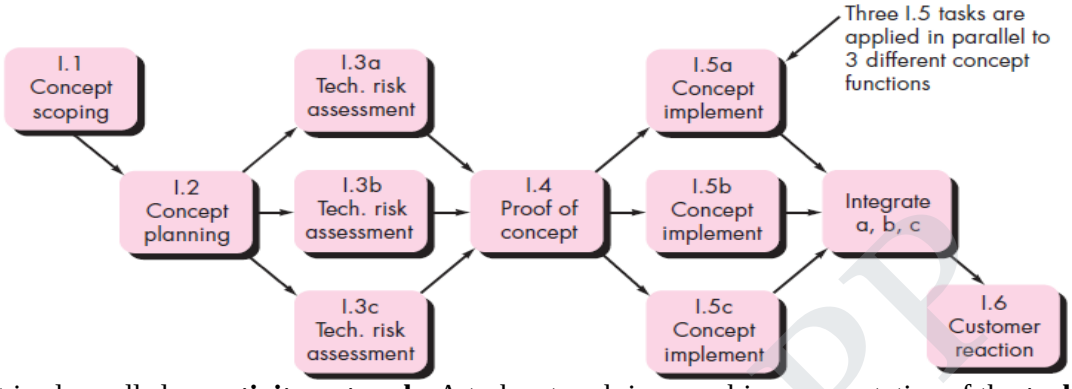
As an example of task decomposition in Task set, consider **Action 1.1**, Concept Scoping. Task refinement can be accomplished using an **outline format**,

```
Task definition: Action 1.1 Concept Scoping
1.1.1   Identify need, benefits and potential customers;
1.1.2   Define desired output/control and input events that drive the application;
        Begin Task 1.1.2
        1.1.2.1 TR: Review written description of need[7]
        1.1.2.2 Derive a list of customer visible outputs/inputs
        1.1.2.3 TR: Review outputs/inputs with customer and revise as required; endtask
        Task 1.1.2
1.1.3   Define the functionality/behavior for each major function;
        Begin Task 1.1.3
        1.1.3.1 TR: Review output and input data objects derived in task 1.1.2;
        1.1.3.2 Derive a model of functions/behaviors;
        1.1.3.3 TR: Review functions/behaviors with customer and revise as required;
        endtask Task 1.1.3
1.1.4   Isolate those elements of the technology to be implemented in software;
1.1.5   Research availability of existing software;
1.1.6   Define technical feasibility;
1.1.7   Make quick estimate of size;
1.1.8   Create a scope definition;
endtask definition: Action 1.1
```

DEFINING A TASK NETWORK

Task Network:

FIGURE 27.2 A task network for concept development



- It is also called an **activity network**. A task network is a graphic representation of the **task flow** for a project.
- **Critical path:**
  - ✓ A single path leading from start to finish in a task network
  - ✓ the tasks on a critical path must be completed on schedule to make the whole project on schedule.
  - ✓ It also determines the minimum duration of the project

SCHEDULING:

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification for software projects.

Program evaluation and review technique (PERT) and the **critical path method (CPM)** are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities such as

1. Estimates of effort
2. A decomposition of the product function
3. The selection of the appropriate process model and task set
4. Decomposition of the tasks that are selected.

Interdependencies among tasks may be defined using a **task network**. Tasks, sometimes called the project **work breakdown structure (WBS)**, are defined for the product as a whole or for individual functions. Both **PERT and CPM provide quantitative tools** that allow you to

(1) determine the critical path—the chain of tasks that determines the duration of the project,
(2) establish "most likely" time estimates for individual tasks by applying statistical models
(3) calculate "boundary times" that define a time "window" for a particular task.

Timeline Chart

- Also called a **Gantt chart;** invented by **Henry Gantt,** industrial engineer, **1917.**
- All project tasks are listed in the far left column. The next few columns may list the following for each task: projected start date, projected stop date, projected duration, actual start date, actual stop date, actual duration, task inter-dependencies (i.e., predecessors)
- To the far right are columns representing dates on a calendar
- The length of a **horizontal bar** on the calendar indicates the duration of the task

- When **multiple bars occur** at the same time interval on the calendar, this implies task **concurrency**
- A **diamond** in the calendar area of a specific task indicates that the task is a **milestone**;

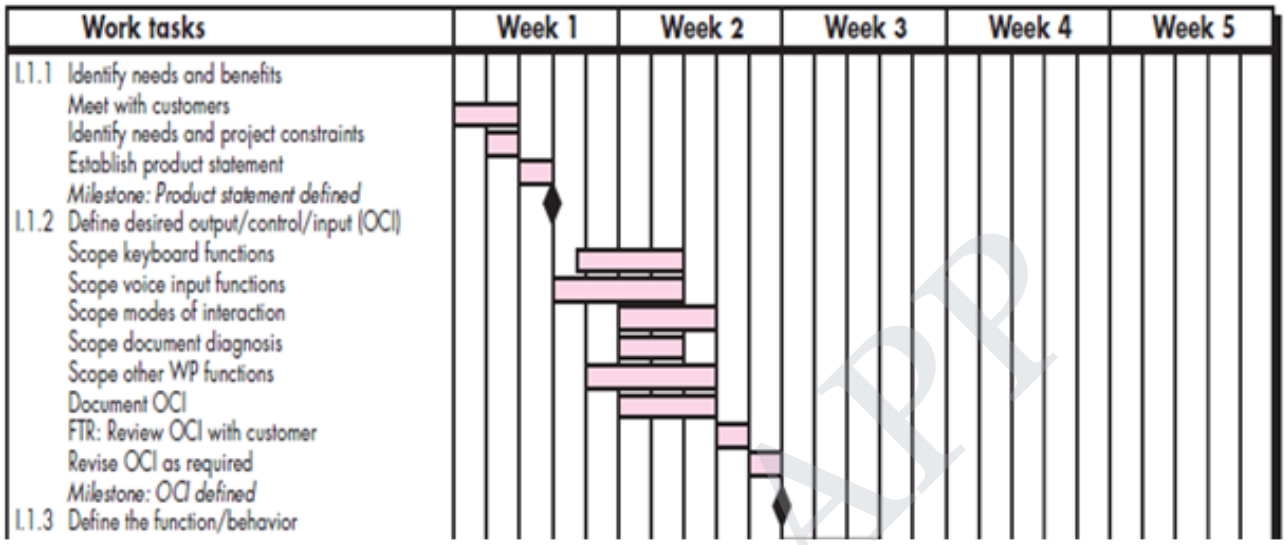


Fig: Timeline Chart

Project Table

Once the information necessary for the generation of a time-line chart has been input, the majority of software project scheduling tools produce **project tables**—a tabular listing of all project tasks, their planned and actual start and end dates, and a variety of related information .

Used in conjunction with the time-line chart, project tables enable you to track progress.

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits							Scoping will require more effort/time
Meet with customers	wk1, d1	wk1, d1	wk1, d2	wk1, d2	BLS	2 p-d	
Identify needs and project constraints	wk1, d2	wk1, d2	wk1, d2	wk1, d2	JPP	1 p-d	
Establish product statement	wk1, d3	wk1, d3	wk1, d3	wk1, d3	BLS/JPP	1 p-d	
Milestone: Product statement defined	wk1, d3	wk1, d3	wk1, d3	wk1, d3			
I.1.2 Define desired output/control/input (OCI)							
Scope keyboard functions	wk1, d4	wk1, d4	wk2, d2		BLS	1.5 p-d	
Scope voice input functions	wk1, d3	wk1, d3	wk2, d2		JPP	2 p-d	
Scope modes of interaction	wk2, d1		wk2, d3		MLL	1 p-d	
Scope document diagnostics	wk2, d1		wk2, d2		BLS	1.5 p-d	
Scope other WP functions	wk1, d4	wk1, d4	wk2, d3		JPP	2 p-d	
Document OCI	wk2, d1		wk2, d3		MLL	3 p-d	
FTR: Review OCI with customer	wk2, d3		wk2, d3		all	3 p-d	
Revise OCI as required	wk2, d4		wk2, d4		all	3 p-d	
Milestone: OCI defined	wk2, d5		wk2, d5				
I.1.3 Define the function/behavior							

Fig: Project Table

Tracking Project Schedule:

If it has been properly developed, the project schedule becomes a road map that defines the tasks and milestones to be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

1. Conducting periodic project status meeting
2. Evaluating the review results
3. Determining whether formal project milestones (the diamonds shown in Figure) have been accomplished by the scheduled date
4. Comparing the actual start date to the planned start date for each project task listed in the project table (Figure)
5. Meeting informally with practitioners to obtain their subjective assessment of progress to date.
6. Using earned value analysis **(EVA)** to assess progress quantitatively**.**

When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called **time-boxing**. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore, an incremental software paradigm is chosen, and a schedule is derived for each incremental delivery. The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A "box" is put around each task. When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

RISK MANAGEMENT

Risk management is the action that help a software team to manage uncertainty.

Reactive verses proactive risk strategies:

Reactive risk strategies: The software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire-fighting mode.

A **proactive strategy** begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk.

SOFTWARE RISK

Risk concerns with failure happenings. Risk involves change, such as in changes in mind, opinion, actions, or places. Risk always involves **two characteristics:**

1. **uncertainty**—the risk may or may not happen;
2. **loss**—if the risk becomes a reality, unwanted consequences or losses will occur

Different categories of risks:

1. **Project risks**
2. **Technical risks**
3. **Business risks-> Market risk ,Strategic risk,Sales risk,Management risk,Budget risks**

1. **Project risks:** threaten the project plan. If project risks become real, it is likely that the project schedule will slip and that costs will increase.

2. **Technical risks:** threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible.
3. **Business risks**: threaten the viability of the software to be built and often risk the project or the product
    i. **Market risk:** building an excellent product or system that no one really wants
    ii. **Strategic risk:** building a product that no longer fits into the overall business strategy for the company
    iii. **Sales risk:** building a product that the sales force doesn't understand how to sell
    iv. **Management risk**: losing the support of senior management due to a change in focus or a change in people
    v. **Budget risks:** losing budgetary or personnel commitment.

Another **categorization of risk** proposed by **Charette** is-

Known risks: Identified after careful evaluation of the **project plan.** Other sources of risk identification are:

- ➔ lack of documented requirements
- ➔ poor development environment
- ➔ unrealistic dead lines

Two types of known risk:

1. **Predictable risks** -identified in advance. Predictable risks are extrapolated from past project experience.
2. **Unpredictable risks** - they are extremely difficult to identify in advance.

ACTIVITIES FOR RISK MANAGEMENT

1. **Risk identification**
2. **Risk projection**
3. **Risk refinement**
4. **Risk mitigation, monitoring and management**

1) Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories:

1. **Generic risks** are a potential threat to every software project.
2. **Product-specific risks** can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.

Risk identification is done by **2 steps**:

Step1: preparation of risk item check list:

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

1. **Product size**—risks associated with the overall size of the software to be built or modified.

2. **Staff size and experience**—risks associated with the overall technical and project experience of the software engineers who will do the work.
3. **Business impact**—risks associated with constraints imposed by management or the marketplace.
4. **Stakeholder characteristics**—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
5. **Process definition**—risks associated with the degree to which the software process has been defined and is followed by the development organization.
6. **Development environment**—risks associated with the availability and quality of the tools to be used to build the product.
7. **Technology to be built**—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

Step2: creating risk component and drivers list:

The project manager identify the risk drivers that affect software risk components— performance, cost, support, and schedule.

1. ***Performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
2. ***Cost risk***—the degree of uncertainty that the project budget will be maintained.
3. ***Support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
4. ***Schedule risk***—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of **four impact levels**

1. negligible
2. marginal
3. critical
4. catastrophic

2)Risk Projection:

Risk projection, also called risk estimation, attempts to rate each risk in **two ways**—

a) the **probability** that the risk is real and
b) the **consequences** of the problems associated with the risk, should it occur.

Four steps for risk projection

1) Establish a **scale** that indicate risk (e.g., 1-low ,10-high)
2) Define the **consequences** of the risk
3) **Estimate the impact** of the risk on the project and product
4) Assess the **overall accuracy** of the risk projection. so that there will be no misunderstandings

These steps help to **prioritize** the risks. Once the risks are prioritized then it becomes easy to allocate the resources for handling them.

Developing a Risk Table

A **risk table** provides you with a simple technique for risk projection. **Steps** for developing a Risk Table are:

1. List **all risks** in the **first column**
2. Mark the **category** of each risk (**7 category**)
3. Estimate the **probability** of each risk occurrence
4. Assess the **impact** of each risk (**4 Impact values**)
5. **RMMM** – Pointer to the **R**isk **M**itigation, **M**onitoring, and **M**anagement

6. **Sort** the rows by probability and impact in **descending order**
7. Draw a **horizontal cutoff line** at some point in the table.
   above cutoff line → first order prioritization
   below cutoff line → second order prioritization

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
Σ				
Σ				
Σ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Fig:Risk Table

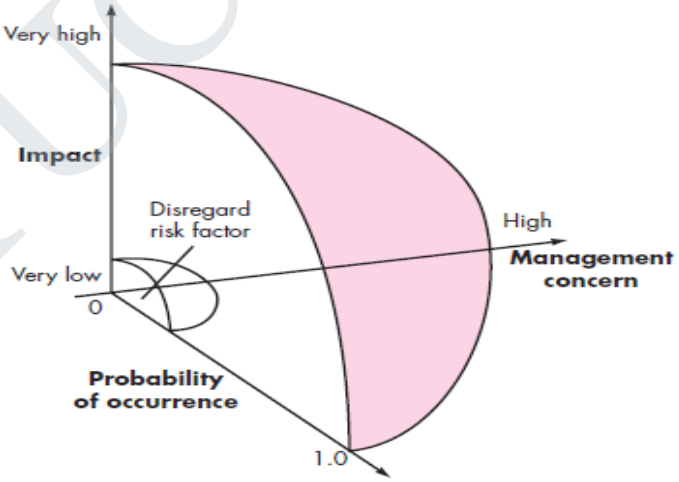


FIGURE 28.3
Risk and management concern

Assessing Risk Impact:

Three factors are considered while assessing risk impact

1. **Its nature** – This indicates **type or kind** of risk
2. **Its scope** – This combines the **severity** of the risk (how serious was it, how much was affected)
3. **Its timing** – This considers **how long** the impact will be felt

Steps to determine the overall consequences of a risk:

(1) determine the average probability of occurrence value for each risk component;
(2) determine the impact for each component based on the criteria
(3) complete the risk table and analyze the results as described in the preceding sections.

The overall **Risk Exposure (RE)** is determined using the following relationship:

RE =P * C

Where, P→probability of occurrence of risk
C→cost

Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made.

For example : Consider a software project with 77 percent of risk probability in which 15 components were developed from the scratch. Each component have on an average 500 LOC and each LOC have an average cost of $10. Then the risk exposure can be calculated as ,

First of all we will compute *cost* = Number of components*LOC*cost of each LOC
= 15*500*10=$75000

Then Risk Exposure = Probability of occurrence of risk × Cost
= 77 / 100 *75000
= $57750

For example, assume that the software team defines a project risk in the following manner:

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. 80 percent (likely).

Risk impact. Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be 18 × 100 × 14 = $25,200.

Risk exposure. RE = 0.80 × 25,200 ~ $20,200.

3)Risk Refinement

One way to do this is to represent the risk in condition-transition-consequence (CTC) format. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

4)Risk Mitigation, Monitoring and Management

An effective strategy must consider **three issues**:

1. **risk avoidance**
2. **risk monitoring**
3. **risk management and contingency planning.**

To mitigate this risk, a strategy has to be developed for reducing turnover. Among the possible steps to be taken are:

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under your control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, **risk-monitoring activities** commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. **Risk management and contingency planning** assumes that mitigation efforts have failed and that the risk has become a reality.

RMMM plan

Risk information sheet			
Risk ID: P02-4-32	Date: 5/9/09	Prob: 80%	Impact: high
Description: Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.			
Refinement/context: Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards. Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components. Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.			
Mitigation/monitoring: 1. Contact third party to determine conformance with design standards. 2. Press for interface standards completion; consider component structure when deciding on interface protocol. 3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.			
Management/contingency plan/trigger: *RE* computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly. Trigger: Mitigation steps unproductive as of 7/1/09.			
Current status: 5/12/09: Mitigation steps initiated.			
Originator: D. Gagne		Assigned: B. Laster	

- The **RMMM plan** documents **all work performed** as part of risk analysis and is **used by** the project **manager** as part of the overall project plan.
- Each risk is documented individually using a ***risk information sheet.***
- Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence.
- **Risk Mitigation** is a problem avoidance activity.
- **Risk Monitoring** is a project tracking activity

SOFTWARE EQUATIONS

(or) Explain "Putnam resources allocation model". Derive the time and effort equations?

The *software equation* is a dynamic multivariable model that assumes a specific distribution of **effort** over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, we derive an estimation model of the form

$$E = \frac{LOC \times B^{0.333}}{P^3} \times \frac{1}{t^4}$$

where

E = effort in person-months or person-years

t = project duration in months or years

B="special skills factor"

P ="productivity parameter" that reflects: overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of the software environment, the skills and experience of the software team, and the complexity of the application

Typical values might be

P =2000 for development of real-time embedded software

P = 10,000 for telecommunication and systems software

P =28,000 for business systems applications.

The productivity parameter can be derived for local conditions using historical data collected from past development efforts. You should note that the software equation has two independent parameters:

(1) an estimate of size (in LOC)

(2) an indication of project duration in calendar months or years.

To simplify the estimation process and use a more common form for their estimation model, Putnam and Myers [Put92] suggest a set of equations derived from the software equation. Minimum **development time** is defined as

$$t_{min} = 8.14\frac{LOC}{P^{0.43}} \text{ in months for } t_{min} > 6 \text{ months}$$

$$E = 180\,Bt^3 \text{ in person-months for } E \geq 20 \text{ person-months}$$

Note that ***t*** in Equation (26.5b) is represented in years. Using Equation (26.5) with P =12,000 (the recommended value for scientific software) for the CAD software.

$$t_{min} = 8.14 \times \frac{33{,}200}{12{,}000^{0.43}} = 12.6 \text{ calendar months}$$

$$E = 180 \times 0.28 \times (1.05)^3 = 58 \text{ person-months}$$

PROJECT PLANNING

The **objective of software project planning** is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded.

Task Set for Project Planning:

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks
4. Define required resources.
   a. Determine required human resources.
   b. Define reusable software resources.
   c. Identify environmental resources.
5. Estimate cost and effort.
   a. Decompose the problem.
   b. Develop two or more estimates using size, function points, process tasks, or use cases.
   c. Reconcile the estimates.
6. Develop a project schedule
   a. Establish a meaningful task set.
   b. Define a task network.
   c. Use scheduling tools to develop a time-line chart.
   d. Define schedule tracking mechanisms.

The **CMMI** defines each **process area** in terms of **"specific goals" and the "specific practices"** required to achieve these goals. ***Specific goals*** establish the characteristics that must exist if the activities implied by a process area are to be effective. ***Specific practices*** refine a goal into a set of process-related activities.

FIGURE 30.3 Process areas required to achieve a maturity level. *Source:* [Phi02].

Level	Focus	Process Areas
Optimizing	*Continuous process improvement*	Organizational innovation and deployment Causal analysis and resolution
Quantitatively managed	*Quantitative management*	Organizational process performance Quantitative project management
Defined	*Process standardization*	Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management Integrated supplier management Risk management Decision analysis and resolution Organizational environment for integration Integrated teaming
Managed	*Basic project management*	Requirements management Project planning Project monitoring and control Supplier agreement management Measurement and analysis Process and product quality assurance Configuration management
Performed		

For example, **project planning** is one of eight process areas defined by the CMMI for "project management" category. The specific goals (SG) and the associated specific practices (SP) defined for **project planning** are

SG 1 Establish Estimates

SP 1.1-1 Estimate the Scope of the Project

SP 1.2-1 Establish Estimates of Work Product and Task Attributes

SP 1.3-1 Define Project Life Cycle

SP 1.4-1 Determine Estimates of Effort and Cost

SG 2 Develop a Project Plan

SP 2.1-1 Establish the Budget and Schedule

SP 2.2-1 Identify Project Risks

SP 2.3-1 Plan for Data Management

SP 2.4-1 Plan for Project Resources

SP 2.5-1 Plan for Needed Knowledge and Skills

SP 2.6-1 Plan Stakeholder Involvement

SP 2.7-1 Establish the Project Plan

SG 3 Obtain Commitment to the Plan

SP 3.1-1 Review Plans That Affect the Project

SP 3.2-1 Reconcile Work and Resource Levels

SP 3.3-1 Obtain Plan Commitment

To illustrate, the generic goals (GG) and practices (GP) for the **project planning** process area are

GG 1 Achieve Specific Goals

GP 1.1 Perform Base Practices

GG 2 Institutionalize a Managed Process

GP 2.1 Establish an Organizational Policy

GP 2.2 Plan the Process

GP 2.3 Provide Resources

GP 2.4 Assign Responsibility

GP 2.5 Train People

GP 2.6 Manage Configurations

GP 2.7 Identify and Involve Relevant Stakeholders

GP 2.8 Monitor and Control the Process

GP 2.9 Objectively Evaluate Adherence

GP 2.10 Review Status with Higher-Level Management

GG 3 Institutionalize a Defined Process

GP 3.1 Establish a Defined Process

GP 3.2 Collect Improvement Information

GG 4 Institutionalize a Quantitatively Managed Process

GP 4.1 Establish Quantitative Objectives for the Process

GP 4.2 Stabilize Subprocess Performance

GG 5 Institutionalize an Optimizing Process

GP 5.1 Ensure Continuous Process Improvement

GP 5.2 Correct Root Causes of Problems

DELPHI METHOD

Under this method of software estimation, the project specifications would be given to a few experts and their opinion taken. The actual number of experts chosen would depend on their availability. A minimum of three is normally selected to have a range of values.

Delphi method has the following steps –

1. Selection of experts
2. Briefing to the experts
3. Collation of estimates from experts
4. Convergence of estimates and finalization

Merits of Delphi technique

1. Very useful when the organization does not have any in-house experts with the domain
2. knowledge or the development platform experience to come out with a quick estimate
3. Very quick to derive an estimate
4. Simple to administer and use

Demerits of Delphi technique

1. This is too simplistic
2. It may be difficult to locate right experts
3. It may also be difficult to locate adequate number of experts willing to participate in the estimation
4. The derived estimate is not auditable
5. It is not possible to determine the causes of variance between the estimated value and the actual values
6. Only size and effort and estimation are possible – schedule would not be available.

Explain the defect removal efficiency and integrity along with its equations.

Integrity:

To measure integrity, two additional attributes must be defined: threat and security. Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as:

$$\text{Integrity} = \sum[1-(\text{threat}*(1-\text{security}))]$$

Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is *defect removal efficiency* (DRE).When considered for a project as a whole, DRE is defined in the following manner:

$$DRE = \frac{E}{E+D}$$

where *E* is the number of errors found before delivery of the software to the end user and *D* is the number of defects found after delivery. DRE can also be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering action.

For example, requirements analysis produces a requirements model that can be reviewed to find and correct errors. Those errors that are not found during the review of the requirements model are passed on to design (where they may or may not be found). When used in this context, we redefine DRE as

$$DRE_j = \frac{E_j}{E_j + D_{j+1}}$$

where *Ei* is the number of errors found during software engineering action *i* and *Ei*+ 1 is the number of errors found during software engineering action *i* + 1 that are traceable to errors that were not discovered in software engineering action *i*. A quality objective for a software team (or an individual software engineer) is to achieve DRE*i* that approaches 1. That is, errors should be filtered out before they are passed on to the next activity or action.

Example:

If team A found 342 errors prior to release of software and Team B found 182 errors. What additional measures and metrics are needed to find out if the teams have removed the errors effectively? Explain.

Solution:

DRE= E/(E+D)

Where

E- Error found before release

D- No.of Defect after release

DRE= 342/(342+182)= 342/524= 0.65

Since DRE is not equal to 1 there exists some defect in the software.

.