CS8494-SOFTWARE ENGINEERING

UNIT-I

1.1. INTRODUCTION TO SOFTWARE ENGINEERING

Software definition:

Definition 1: Software is instructions (computer programs) that are intended to provide desired Features, function, and performance;

Definition 2: Software is a data structure that enables the programs to adequately manipulate information.

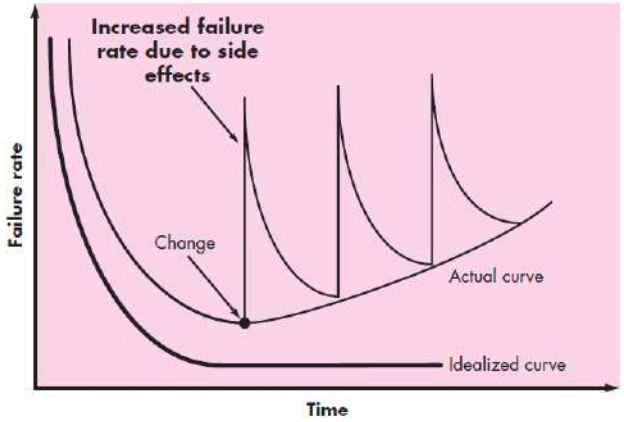
Characteristics of software are

1. **Software is developed or engineered; it is not manufactured in the classical sense.**

   In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software

2. **Software doesn't "wear out."**

   The failure rate curve for software should take the form of the "idealized curve" shown in Figure. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate!



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

Software Application Domains

Seven broad categories of computer software

1) **System software**—a collection of programs written to service other programs. Some system software are compilers, editors, and assembler. The purpose of the sysem software is to establish a communication with the hardware.
2) **Application software**—stand-alone programs that solve a specific business need.
3) **Engineering/scientific software**—has been characterized by "number crunching" algorithms.
4) **Embedded software**—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself.
5) **Product-line software**—designed to provide a specific capability for use by many different customers.
6) **Web applications**—called "WebApps," this network-centric software category spans a wide array of applications.
7) **Artificial intelligence software**—makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.

Legacy software systems:

Legacy software systems . . . were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms.

A few simple realities to build software that is ready to meet the challenges of the twenty-first century are:

1. A concerted effort should be made to understand the problem before a software solution is developed.
2. Design becomes a pivotal activity
3. Software should exhibit high quality
4. Software should be maintainable

Software in all of its forms and across all of its application domains should be engineered.

Software engineering:

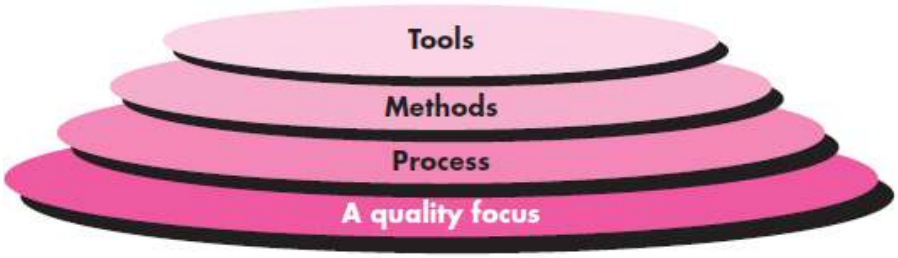
A definition proposed by Fritz Bauer is

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

The IEEE definition is:

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

Software engineering is a layered technology.



1) **Quality focus**

A disciplined quality management is a backbone of software engineering technology.

2) **Process layer:**

The foundation for software engineering is the *process* layer. Process defines a framework that must be established for effective delivery of software engineering technology.

3) **Methods:**

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.

4) **Tools:**

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

1.2. SOFTWARE PROCESS:

- A **process** is a collection of activities, actions, and tasks that are performed when some work product is to be created.
- An **activity** strives to achieve a broad objective (e.g., communication with stakeholders) 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.
- An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

- **A task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.
- In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

A **process framework** establishes the foundation for a complete software engineering process by identifying a small number of **framework activities** that are applicable to all software projects, regardless of their size or complexity.

A generic process framework for software engineering encompasses five activities:

1) **Communication:**

Before any technical work can commence, it is critically important to communicate and collaborate with the customer. The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

2) **Planning:**

software project plan—defines the software engineering work by describing 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.

3) **Modeling:**

Software engineers will create models to better understand software requirements and the design that will achieve those requirements.

4) **Construction:**

This activity combines code generation and the testing that is required uncovering errors in the code.

5) **Deployment:**

The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems.

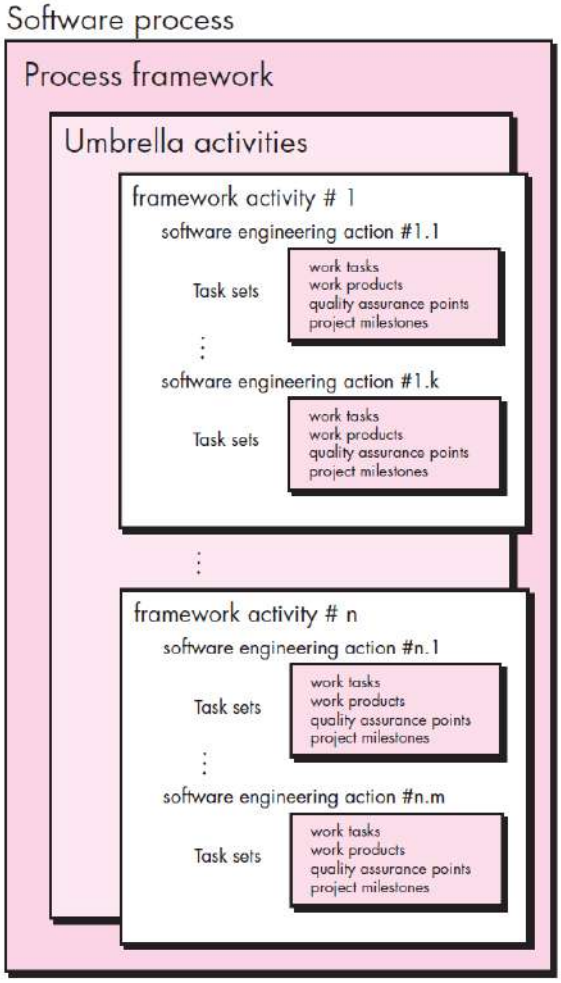
Umbrella activities:

Umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

Typical umbrella activities include:

1) **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
2) **Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.
3) **Software quality assurance**—defines and conducts the activities required to ensure software quality.
4) **A technical review—assesses software** engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
5) **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs.
6) **Software configuration management**—manages the effects of change throughout the software process.
7) **Reusability management**—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

8) Work product preparation and production—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.



1.3.PRESCRIPTIVE PROCESS MODELS (OR) LIFE CYCLE MODELS:

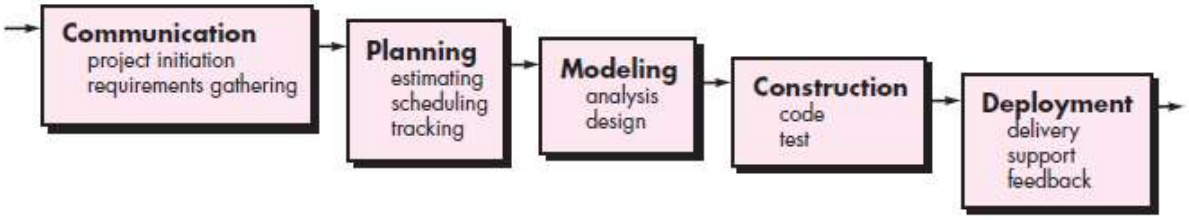
The process model can be defined as the abstract representation of process. The appropriate process model can be chosen based on abstract representation of process. These process models will follow some rules for correct usage.

It is called "prescriptive" model because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

1.3.1.The Waterfall Model (or)classic life cycle (or) sequential life cycle model

(or) Software Development Life Cycle (SDLC)

(or) Systems development life cycle (SDLC)



- The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.
- The waterfall model is the oldest paradigm for software engineering.
- **In requirement gathering and analysis phase** the basic requirements of the system must be understood by software engineer, who is called analyst.
- The **design** is an intermediate step between requirements analysis and coding.
  Design focuses on:
  1) Data Structure
  2) Software architecture
  3) Interface representation
  4) Algorithm details
- **Coding** is a step in which design is translated into **machine readable form**.
- **Testing** begins when coding is done. The purpose of testing is to uncover errors, fix the bugs and meet the customer requirements**.**
- **Maintenance** is the **longest life cycle** phase. The purpose of maintenance is when the system is installed and put in practical use then error may get introduced, correcting such errors and **putting it in use.**

Advantages:

1) The waterfall model is simple to implement
2) For implementation of small systems it is usefull.

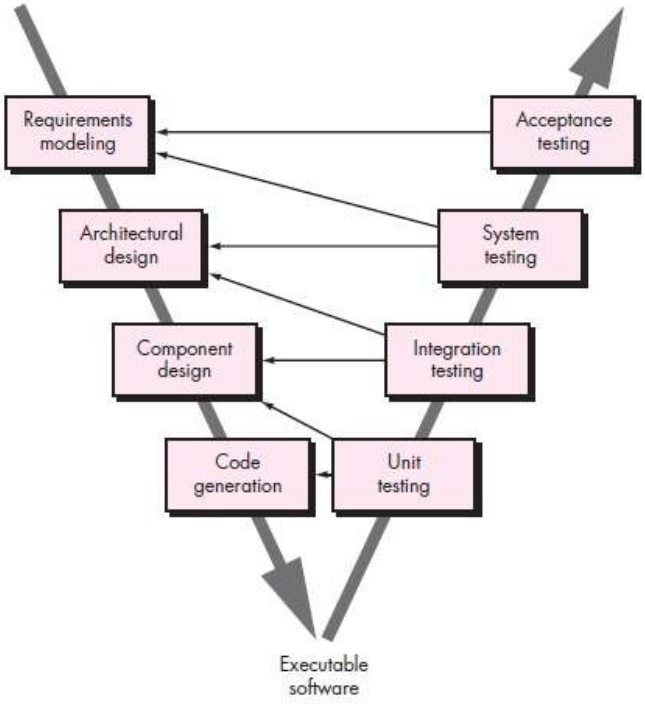
Problems in waterfall model:

1. Real projects rarely follow the sequential flow that the model proposes. Changes can cause confusion as the project team proceeds.

2. It is difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

V-Model: In each phase, testing will be done.



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

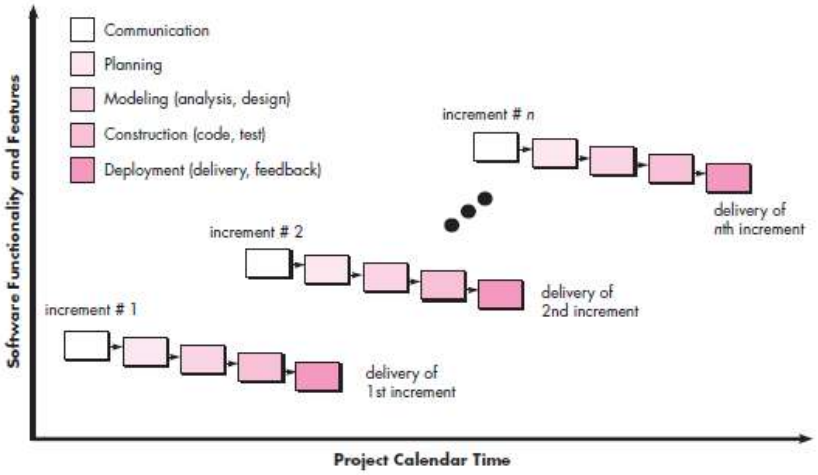
- The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities. As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations 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.
- In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

1.3.2. Incremental Process Models

- The incremental model **combines elements of linear and parallel process flows**.
- The incremental model delivers series of releases to the customer. These releases are called **increments**. More and more functionality is associated with each increment.
- The incremental model combines elements of linear and parallel process flows. The incremental model applies linear sequences in a staggered fashion as calendar time progresses.
- Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.

When we can choose incremental:

1) When initial software requirements are reasonably well defined
2) When the overall scope of the development effort precludes a purely linear process.
3) When limited set of software functionality needed quickly



- The incremental model applies linear sequences in a staggered fashion as **calendar time progresses.**
- **For example, word-processing** software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; Spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
- The **first increment** is often a **core product**. That is, basic requirements are addressed but many supplementary features remain undelivered.
- The core product is used by the customer. As a result of use, 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 following the delivery of each increment, until the complete product is produced.
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people.
- If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.

Advantages:

1) Generates working software quickly and early during the software life cycle.
2) This model is more flexible – less costly to change scope and requirements.
3) It is easier to test and debug during a smaller iteration.
4) In this model customer can respond to each built.
5) Lowers initial delivery cost.
6) Easier to manage risk because risky pieces are identified and handled during it'd iteration.

Disadvantages:

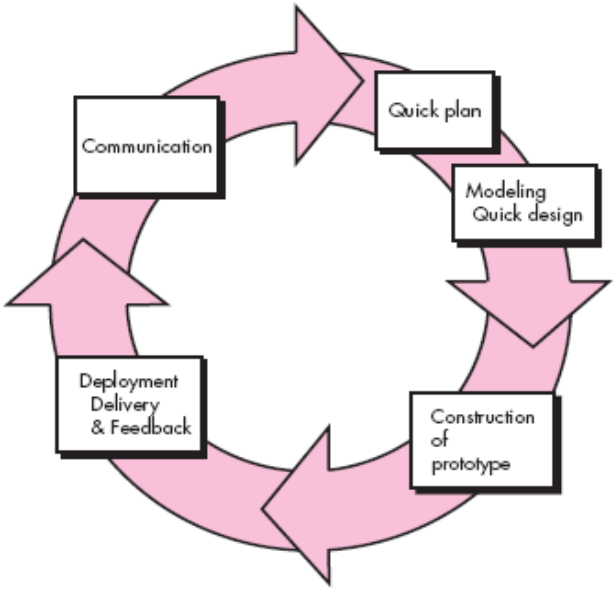
1) Needs good planning and design.
2) Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
3) Total cost is higher than waterfall.

1.3.3. Evolutionary Process Models

- Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; In such case, the iterative approach needs to be adopted. Evolutionary process model is also called as iterative process model
- Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

1.3.3.1.Prototyping

- **Software prototyping**, refers to the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.
- **When we can choose Prototype:**
  - A customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features.
  - The developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system
  - When requirements are fuzzy
- Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models.
- Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models.
- The prototyping assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.
- The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).



- The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.

- Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done
- Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.
- In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.
- The prototype can serve as "the first system." The one that Brooks recommends you throw away. But this may be an idealized view. Although some prototypes are built as "throwaways," others are evolutionary in the sense that the prototype slowly evolves into the actual system.
- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately.

Advantages:

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

Disadvantages:

1) Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.
2) Software engineer make implementation compromises in order to get a prototype working quickly.
3) An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability.

Usage of prototyping:

- Although problems can occur, prototyping can be an effective paradigm for software

Engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

1.3.3.2.The Spiral Model.

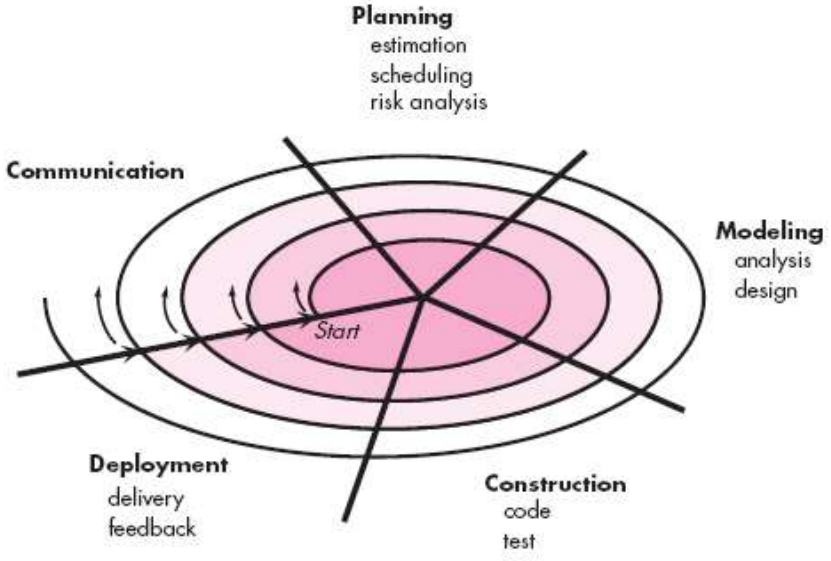
- The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software.
- The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.

It has two main distinguishing features.

(1) One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.

(2) The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

- A spiral model is divided into a set of framework activities defined by the software engineering team. Each of the framework activities represent one segment of the spiral path.
- The spiral model is a realistic approach to the **development of large-scale systems and software**. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.
- The spiral model uses prototyping as a **risk reduction mechanism** but enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
- The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.



The functions of these four quadrants are discussed below-

- **Objectives determination and identify alternative solutions (Concept development projects):** Requirements are gathered from the customers and the objectives are identified, elaborated and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.
- **Identify and resolve Risks (New product development projects):** During the second quadrant all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution is identified and the risks are resolved using the best possible strategy. At the end of this quadrant, Prototype is built for the best possible solution.
- **Develop next version of the Product (Product Enhancement projects):** During the third quadrant, the identified features are developed and verified through testing. At the end of the third quadrant, the next version of the software is available.
- **Review and plan for the next Phase (product Maintenance projects):** In the fourth quadrant, the Customers evaluate the so far developed version of the software. In the end, planning for the next phase is started.

Advantages:

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.

- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.

Disadvantages:

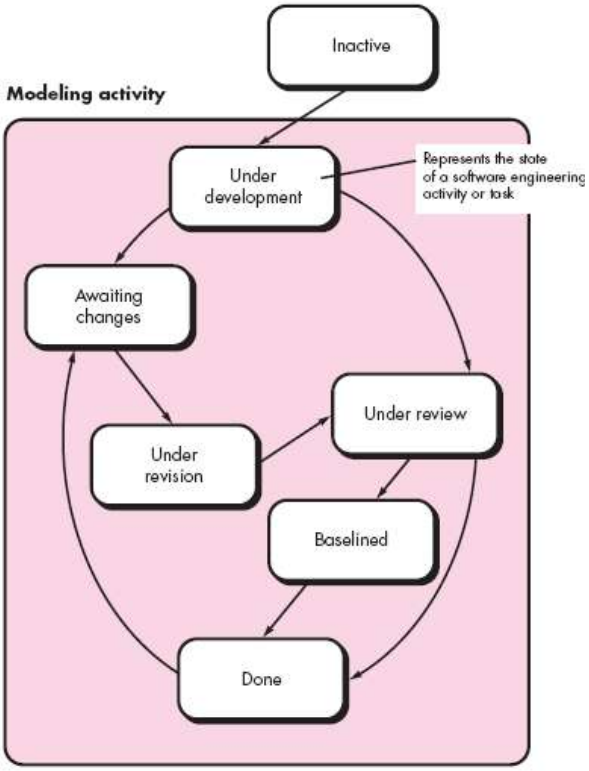
- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

1.3.4. Concurrent development Models

- The concurrent development model, sometimes called concurrent engineering, allows a software team to represent iterative and concurrent elements of any of the process models.
- For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following software engineering actions: prototyping, analysis, and design.
- Figure provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach. The activity—**modeling**—may be in any one of the states noted at any given time.
- Similarly, other activities, actions, or tasks (e.g., **communication** or **construction**) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.
- For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the **awaiting changes** state. The modeling activity (which existed in the **inactive** state while initial communication was completed, now makes a transition into the **under development** state.
- If the customer indicates that changes in requirements must be made, the modeling activity moves from the **under development** state into the **awaiting changes** state.
- Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.
- For example, during early stages of design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model is uncovered. This generates the event analysis model correction, which will trigger the requirements analysis action from the **done** state into the **awaiting changes** state.
- Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a process network.
- Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks. Events generated at one point in the process network trigger transitions among the states.

Advantages:

1) The concurrent development model, sometimes called concurrent engineering. It's can be represented schematically as a series of frame work activities, software engineering actions, software engineering task and their associated states.
2) The concurrent process model defines a series of events that will trigger transition from state to state for each of the software engineering activities and action or task.
3) The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.



Disadvantages:

1) The SRS must be continually updated to reflect changes.
2) It requires discipline to avoid adding too many new features too late in the project.

1.4. SPECIALIZED PROCESS MODELS

- Specialized process models take on many of the characteristics of one or more of the traditional models. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

1.4.1. Component-Based Development

- Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software.
- However, the component-based development model constructs applications from prepackaged software components.
- Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages16 of classes.
- Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1) Available component-based products are researched and evaluated for the application domain in question.
2) Component integration issues are considered.
3) Software architecture is designed to accommodate the components.

4) Components are integrated into the architecture.
5) Comprehensive testing is conducted to ensure proper functionality.

- The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture.

Advantages:

- The component based development model leads to software re-used and re-usability provides a number of tangible benefits.
- It leads to reduction in development cycle time.
- It leads to significant reduction in project cost.
- It leads to significant increase in productivity.

Disadvantages:

1) **Customization**
2) Problem to **adapt** a component
3) **The integration** of a reusable component into new component is also a major problem
4) **Security** is another major concern for the developers
5) **Efficiency** of the Software applications developed using CBD is also debatable.

1.4.2. The Formal Methods Model

- The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called clean room software engineering is currently applied by some software development organizations.
- When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms.
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Advantage:

- Although not a mainstream approach, the formal methods model offers the promise of defect-free software.

Disadvantages:

1) The development of formal models is currently quite time consuming and expensive.
2) Because few software developers have the necessary background to apply formal methods, extensive training is required.
3) It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

1.4.3. Aspect-Oriented Software Development

- Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture.
- As modern computer-based systems become more sophisticated (and complex), certain concerns—customer required properties or areas of technical interest—span the entire architecture.

- Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).
- When concerns cut across multiple system functions, features, and information, they are often referred to as crosscutting concerns. Aspectual requirements define those crosscutting concerns that have an impact across the software architecture.
- Aspect-oriented software development (AOSD), **often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects**—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern"
- A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed.
- The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

COMPARISON OF DIFFERENT SDLC MODELS

Waterfall Model	Spiral Model	Prototyping Model	Incremental Model
Requirements must be clearly understood and defines at the beginning only.	The **requirements** analysis and gathering can be done in iteration because requirements get changed quite often.	**Requirements analysis** can be made in the later stages of the development cycle. Because requirements get changed quite often.	**Requirements analysis** can be made in the later stages of the development cycle.
The **development team** having the adequate experience of working on the similar project is chose to work on this type of process model	The **development team** having the adequate experience of working on the similar project is allowed in this process model.	The **development team** having the adequate experience of working on the similar project is allowed in this process model.	The **development team** having the adequate experience of working on the similar project is chose to work on this type of process model
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.
When the requirements are reasonably well defined and the development effort suggests a purely linear effort then the **waterfall model is chosen.**	Due to iterative nature of this model, the risk identification and rectification is done before they get problematic. Hence for handling real time problems the **spiral model is chosen.**	When developer is unsure about the efficiency of an algorithm or the adaptability of an operating system then the **prototyping model is chosen**.	When the requirements are reasonably well defined and the development effort suggests a purely linear effort and when limited set of software functionality is needed quickly then the **incremental model is chosen**.

The comparison of the different models is represented in the following table on the basis of certain features.

FEATURES	WATERFALL	V-SHAPED	INCREMENTAL	SPIRAL	RAD
Requirement specifications	Beginning	Beginning	Beginning	Beginning	Time boxed release
Cost	Low	Expensive	Low	Expensive	Low
Simplicity	Simple	Intermediate	Intermediate	Intermediate	Very Simple
Risk involvement	high	Low	Easily manageable	Low	Very low
Expertise	High	Medium	High	High	Medium
Flexibility to change	Difficult	Difficult	Easy	Easy	Easy
User involvement	Only at beginning	At the beginning	Intermediate	High	Only at the beginning
Flexibility	Rigid	Little flexible	Less flexible	flexible	High
Maintenance	Least	Least	Promotes maintainability	Typical	Easily maintained
Duration	Long	According to project size	Very long	Long	Short

Table 1(Comparison of Different SDLC Models)

1.5 Introduction to Agility:

Agile is a time-bound, iterative approach to software delivery that builds software incrementally from the start of the project, instead of trying to deliver all at once.

The **agile manifesto** for agile software development is a formal declaration of **four values and 12 principles** to guide an iterative and people centric approach to software development

Individuals and interactions	over	Process and Tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contact negotiation
Responding to changes	over	Following a Plan

Fig. Agile Manifesto

Why Agile?

Technology in this current era is progressing faster than ever, enforcing the global software companies to work in a fast-paced changing environment. Because these businesses are operating in an ever-changing environment, it is impossible to gather a complete and exhaustive set of software requirements. Without these requirements, it becomes practically hard for any conventional software model to work.

Agile was specially designed to meets the needs of the rapidly changing environment by embracing the idea of incremental development and develop the actual final product.

1.6. Agile Process:

- In 1980's the **heavy weight, plan based** software development approach was used to develop any software product.

- In this approach too many things are done which were not directly related to software product being produced.
- If requirements get changed, then rework was essential. Hence new methods were proposed in 1990's which are known as agile process.
- The agile process is light-weight methods which are **people-based** rather than plan-based methods.
- The agile process forces the development team to **focus on software** itself rather than design and documentation.
- The agile process believes in **iterative method**.
- The aim of agile process is to **deliver** the working model of software **quickly** to the customer.

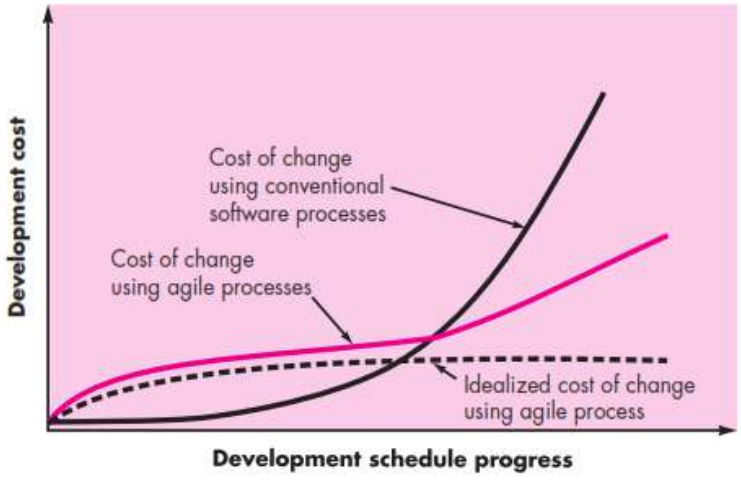
Conventional software Development Methodology:

- The conventional wisdom in software development is that the cost of change increases nonlinearly as a project progresses.
- It is relatively easy to accommodate a change when a software team is gathering requirements. A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited.
- As the progresses and if the customer suggest the changes during the testing phase of the SDLC then to accommodate these changes the architectural design needs to be modified and ultimately these changes will affect other phases of SDLC. These changes are actually costly to execute**.**

Agile Methodology:

When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming then the cost of changes can be controlled.

The following graph represents the how the software development approach has a strong influence on the development cost due to changes suggested.



1.6.1Principles:

There are famous 12 principles used as agile principles:

1. Highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. It welcomes changing requirements, even late in development.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shortest timescale.
4. Business people and developers must work together throughout the project.
5. Build projects around motivated individuals. Give them the environment and the 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 constant development. The sponsors, developers, and users should be able to maintain a constant.
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 team must be self– organizing teams for getting best architectures, requirements, and designs emerge from
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

1.7 Extreme programming:

Extreme programming (XP) is one of the best known agile processes.

1.7.1 XP values:

The set of five *values* that serve as a basis for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

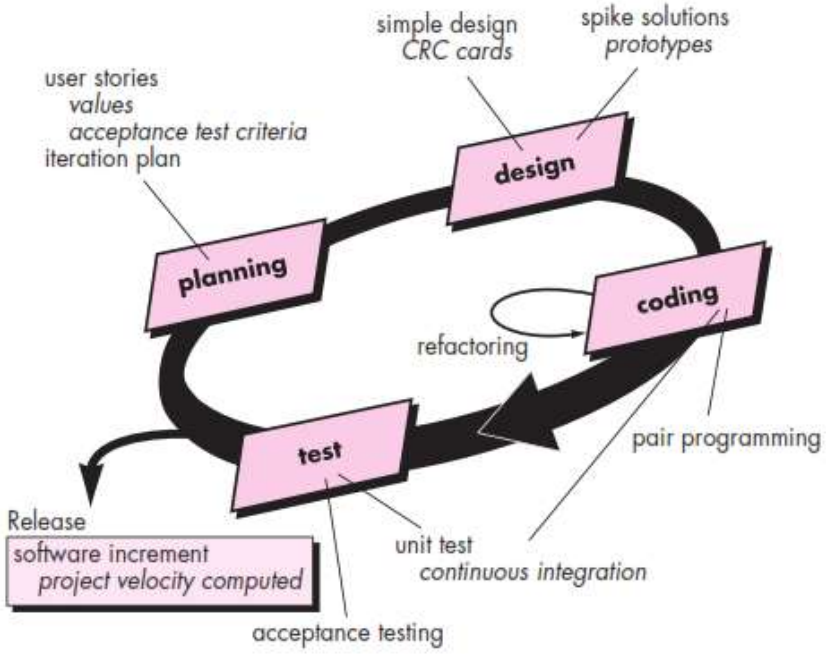
1. **Communication:**
   To achieve effective *communication* between software engineers in order to covey important concepts and to get continuous feedback.

2. **Simplicity:**
   XP focuses on the current needs instead of future needs to incorporate in the design. Hence the XP believes that the Software design should be simple.
3. **Feedback:**
   The feedback for the software product can be obtained from the developers of the software, customers and other software team members.
4. **Courage:**
   the strict adherence to certain XP practices require courage. The agile XP team must be disciplined to design the system today, recognize the future requirements and make the changes dramatically as per demand.
5. **Respect**:
   By following the above states XP values the agile team can win the respect of the stakeholders.

1.7.2 Process:

The extreme programming process is explained as follows -

- Customer specifies and priorities the system requirements. Customer becomes or of the important members of development team. The developer and customer together prepare a **story-card** in which customer needs are mentioned.
- The developer team then aims to implement the scenarios in the story-card.
- After developing the story-card the development team breaks down the total work in **small tasks**. The efforts and the estimated resources required for these tasks are estimated.
- The customer priorities the stories for implementation. If the requirement changes then sometimes unimplemented stories have to be discarded. Then release the complete software in **small** and frequent **releases**.
- For accommodating new changes, **new story-card** must be developed.
- Evaluate the system along with the customer.



Various rules and practices used in extreme programming are as given below-

	XP Principle	Description
Planning	**User story-cards**	Instead of creating a large requirement document user stories are written by the customer in which what they need is mentioned.
	Release planning	A release plan for overall project is prepared from which the iteration plan can be prepared for individual iteration
	Small releases	The developer breaks down the user Stories into small releases and a plan for releasing the small functionalities is prepared.
	Iterative process	Divide the development work into small iterations. Keep the iteration of nearly constant length. Iterative development helps in quick or agile development.
	Stand up meetings	The stand up meetings must be, conducted for the current outcomes of the project.
Designing	**Simple design**	Simple design always takes less time than the complex design. It is always good to keep the things simple to meet the current requirements
	Spike solution	For answering the tough technical problems create the spike solutions. The goal of these solutions should be to reduce the technical risks.
	Refactoring	Refactoring means reductions in the redundancy, elimination of unused functionalities, redesign the obsolete designs. This will improve the quality of the project.
Coding	**Customer availability**	The most essential requirement of the XP is availability of the customer. In Extreme programming the customer not only helps the developer team but it should be the part of the project.

	Paired programming	All the code to be included in the project must be coded by groups of two people working at the same computer. This will increase the quality of coding
	Collective code ownership	By having collective code ownership approach the everyone contributes new ideas and not any single person becomes the bottleneck of the project. Anyone can change any line of code to fix a bug or to refactor.
Testing	**Unit testing**	The test framework that contains automated test case suite is used to the code. All the code must be using unit testing before its release.
	Continuous integration	As soon as one task is finished integrate it into the whole system. Again after such integration unit testing must be conducted
	No overtime	Working overtime loses the spirit and motivation of the team. Conduct the release of the team. Conduct the release planning meeting to change the project scope or to reschedule the project

Applications of Extreme Programming (XP): Some of the projects that are suitable to develop using XP model are given below:

- **Small projects:** XP model is very useful in small projects consisting of small teams as face to face meeting is easier to achieve.
- **Projects involving new technology or Research projects:** This type of projects face changing of requirements rapidly and technical problems. So XP model is used to complete this type of projects.

1.7.3. Industrial XP:

The industrial XP (IXP) is an organic evolution of XP. It is customer-centric. It has expanded role for customers, and its advanced technical practices.

Various new practices that are appended to XP to create IXP are as follows:

1. **Readiness Assessment:**
   Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.*
   (1) an appropriate development environment exists to support IXP
   (2) The team should contain appropriate and skilled stakeholders
   (3) The organization has a distinct quality program and supports continuous improvement
   (4) The organizational culture will support the new values of an agile team.
   (5) The broader project community will be populated appropriately.
2. **Project Community:**
   Skilled and efficient people must be chosen as the agile team members for the success of the project. The team is reffered as the community when extreme programming approach is considered. The project community consists of technologies, customers, and other stakeholders

who play rthe vital role for the success of the project. The role of the community members must be explicitly defined

3. **Project charting:**
   Project charting means assessing the justification for the project as a business application. That means, the IXP team assess whether the project satisfies the goals and objectives of the organization.
4. **Test driven management:**
   For assessing the state of the project and its progress the industrial XP needs some measurable criteria. In test driven management the project is tested with the help of these measurable criteria.
5. **Retrospectives:**
   After delivering the software increment, the specialized review is conducted which is called as retrospective. The intention of retrospective is to improve the industrial XP process.

1.8 CMMI:

The Capability Maturity Model Integration (CMMI) is a capability maturity model developed by the Software Engineering Institute, part of Carnegie Mellon University in Pittsburgh, USA. The CMMI principal is that "the quality of a system or product is highly influenced by the process used to develop and maintain it". CMMI can be used to guide process improvement across a project, a division, or an entire organization.

CMMI provides:

- Guidelines for processes improvement
- An integrated approach to process improvement
- Embedding process improvements into a state of business as usual
- A phased approach to introducing improvements

CMMI Models

CMMI consists of three overlapping disciplines (constellations) providing specific focus into the Development, Acquisition and Service Management domains respectively:

- CMMI for Development (CMMI-DEV) – Product and service development
- CMMI for Services (CMMI-SVC) – Service establishment, management, and delivery
- CMMI for Acquisition (CMMI-ACQ) – Product and service acquisition

Originating in software engineering, CMMI has been highly generalised over the years to embrace other business processes such as the development of hardware products, service delivery and purchasing which has had the effect of abstracting CMMI.

ANNA UNIVERSITY QUESTIONS

PART A

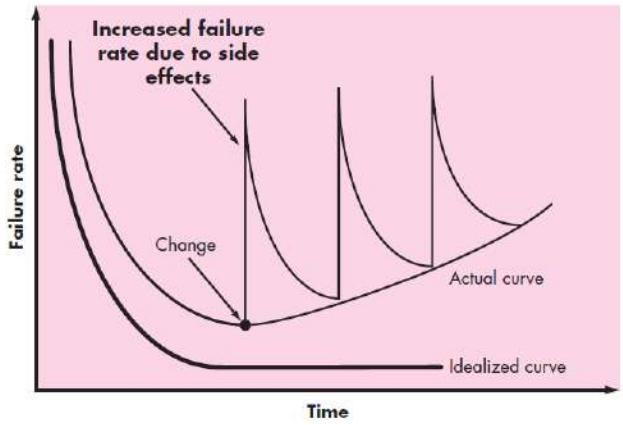
1. **What is Software Engineering? NOV/DEC 2013, NOV / DEC 2014, APRIL/MAY 2017, APRIL/MAY 2017**

   Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software as well as the study of approaches of the same.

2. **'Software doesn't wear out'. Justify. NOV/DEC 2013, MAY/JUNE 2016**

The failure rate curve for software should take the form of the "idealized curve" shown in Figure. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate!

Figure : Failure curves for software



3. **Differentiate : Verification Vs Validation. NOV / DEC 2014**

Verification	Validation
The set of activities that ensure that software correctly implements a specific function.	The set of activities that ensure that the software has been built is traceable to customer requirements
Verification represents the set of activities that are carried out to confirm that the software correctly implements the specific functionality	Validation represents the set of activities that ensure that the software that has been built is satisfying the customer requirements.

4. **Write the Process framework and Umbrella activities. APRIL/MAY 2015**

   - Software project tracking and control.
   - Risk management.
   - Software Quality Assurance.
   - Formal Technical Reviews.
   - Software Configuration Management.
   - Work product preparation and production.
   - Reusability management.
   - Measurement.

5. **What are the pros and cons of Iterative software development models? NOV/DEC 2015**

   The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

6. **If you have to develop a word processing software product, what process model will you choose? Justify your answer. NOV/DEC 2016**

   Incremental model: incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; Spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment

7. **Depict the relationship between Work product, task, activity and System. NOV/DEC 2016, APRIL/MAY 2017**

   - Each framework activity under umbrella activities of software process framework consists of various task set.
   - Each task set consists of work task, work products, quality assurance points and project milestones. The task also accommodates the needs of the system getting developed.

8. **List two deficiencies in waterfall model. Which process model do you suggest to overcome each deficiency? APRIL/MAY 2017**

   i. It is difficult to define all the requirements at the beginning of project, this model is not suitable for accommodating any changes.

   To overcome this efficiency: prototyping

   ii. It does not scale up to large project

   To overcome this efficiency: spiral model.

9. **What is software? List the characteristics**. **APRIL/MAY 2018**

   Software is instructions (computer programs) that are intended to provide desired Features, function, and performance

   Characteristics:

   - Software is developed or engineered;
   - Software doesn't "wear out"
   - most software continues to be custom built.

10. **How does "project Risk" factor affect the spiral model of software development?**

    The spiral model demands considerable risk assessment because if a major risk is not uncovers and managed, problems will occur in the project and then it will not be acceptable by end user.

11. **What led to the transition from product oriented development to process oriented development to process oriented development? APRIL/MAY 2016**

    The software process model led to the transition from product oriented development to process oriented development.

12. **What is the significance of the spiral model when compared with other models. NOV/DEC 2017**

    1. High amount of risk analysis.
    2. Good for large and mission-critical projects.
    3. Software is produced early in the software life cycle.

13. **Write a note on the unique characters of software. NOV/DEC 2017**

    - Functionality
    - Reliability
    - Usability
    - Efficiency

- Maintainability
- Portability

14. What is software process?

The process model can be defined as the abstract representation of process. The appropriate process model can be chosen based on abstract representation of process. These process models will follow some rules for correct usage.

15. Define an evolutionary prototype.

- Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.
- **Software prototyping**, refers to the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed.

16. List any two agile process model.

The following are the agile process model.

- Extreme Processing (XP)
- Agile Modeling
- Scrum

PART B

1. Compare the following life cycle models based on their distinguishing factors, strengths and weaknesses — Waterfall Model, RAD Model, Spiral Model and Formal Methods Model. (Present in the form of table only — use diagrams wherever necessary) **NOV/DEC 2013, NOV/DEC 2018**
2. Assume that you are the technical manager of a software development organization. A client approached you for a software solution. The problems stated by the client have uncertainties which lead to loss if it not planned and solved. Which software development model you will suggest for this project – Justify. Explain that model with its pros and cons and neat sketch. **NOV/DEC 2015**
3. Explain the various levels of capability maturity model integration. **NOV/DEC 2015**
4. Discuss the prototyping model. What is the effect of designing a prototype on the overall cost of the software project? **MAY/JUNE 2016**
5. Describe the type of situations where iterative enhancement model might lead to difficulties **MAY/JUNE 2016**
6. Elucidate the key features of the software process models with suitable examples. **MAY/JUNE 2016**
7. What is the role of user participation in the selection of a life cycle model? **MAY/JUNE 2016**
8. Which process model is best suited for risk management? Discuss in detail with an example. Give the advantages and disadvantages of the model. **NOV/DEC 2016**
9. List the principles of agile software development. **NOV/DEC 2016, NOV/DEC 2019**
10. What is a process model? Describe the process model that you would choose to manufacture a car. Explain giving suitable reasons. **MAY/JUNE 2017, NOV/DEC 2019 (Spiral Model)**

QUESTION BANK

PART A

1. What is the impact of reusability in software development process?
2. Explain the component based software development model with a neat sketch.
3. What are the characteristics of software?
4. Software doesn't wears out. Justify
5. What are the layers of software engineering?
6. Write down the generic process framework that is applicable to any software project.
7. List the goals of Software Engineering

8. Give two reasons why system engineers must understand the environment of a system.
9. What are the two types of software products?
10. What is Software Engineering? What are their applications?
11. List out evolutionary software process model.
11. What are the difference between product and process?
12. What are the advantages and disadvantages of Waterfall Model?
13. What are the advantages and disadvantage of Incremental Model?
14. What are the advantages and disadvantages of Spiral Model?
15. Define Computer based system and specify its components.
16. What are the advantages and disadvantages of Prototyping Model?
17. Depict the relationship between work product, task, activity and system.
18. List two deficiencies in waterfall model. Which process model do you suggest to overcome each deficiency?
19. What are the pros and cons of Iterative software development models?
20. What is legacy software? What are characteristics of legacy software?
21. What is an agile process?
22. List out the principles of Agile.

PART B

1. Explain the following: (i) waterfall model (ii) Spiral model (iii) Prototyping model
2. Discuss the various life cycle models in software development.
3. Compare and contrast the different lifecycle models.
4. Discuss in detail about any two evolutionary process models.
5. Explain in detail about the software process.
6. Which process model is best suit for risk management? Discuss in detail with an example. Give advantages and disadvantages of the model.
7. List the principles of agile development.
8. Explain in details about extreme programming XP process.
9. Explain about agile process.

CS6403 / SOFTWARE ENGINEERING

UNIT II- REQUIREMENTS ANALYSIS AND SPECIFICATION

SYLLABUS:

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.

2.1. SOFTWARE REQUIREMENTS:

- The process of finding out, analyzing, documenting and checking these services and constraints is called **Requirements engineering (RE).**
- 'User requirements' to mean the high-level abstract requirements and 'system requirements' to mean the detailed description of what the system should do.

1. **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.

2. **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 of the contract between the system buyer and the software developers.

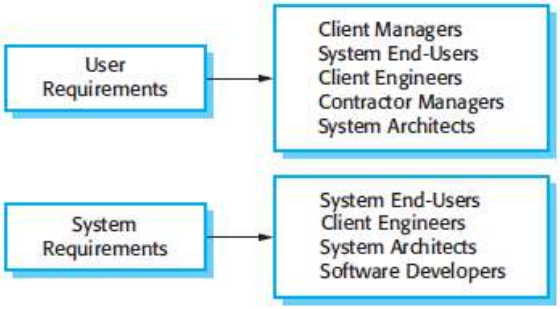


Figure. Readers of different types of requirements specification

Software system requirements are classified as functional requirements, nonfunctional requirements and domain requirements:

1. **Functional requirements:**

These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. **Non-functional requirements**

These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.

3. **Domain requirements**

These are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain. They may be functional or non-functional requirements

2.2. FUNCTIONAL REQUIREMENTS:

- The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.
- When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users.
- More specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.
- The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high level functions {*fi*}. The functional view of the system is shown in fig. Each function fi of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data (*oi*).

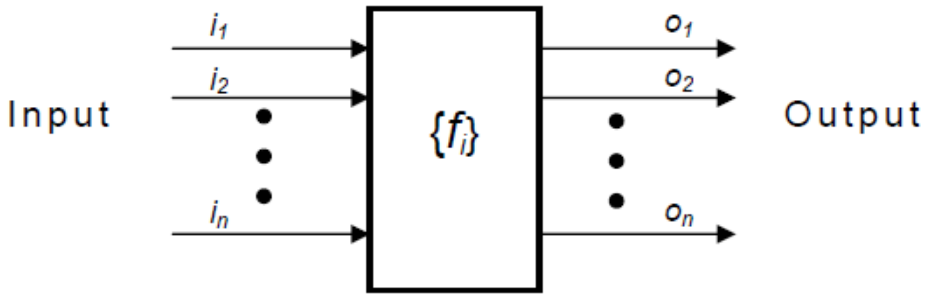


Fig: View of a system performing a set of functions

- The user can get some meaningful piece of work done using a high-level function.
- The functional requirements specification of a system should be both complete and consistent.
- Completeness means that all services required by the user should be defined.
- Consistency means that requirements should not have contradictory definitions.
- In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness.

Reasons are:

1. It is easy to make mistakes and omissions when writing specifications for complex systems.
2. There are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different— and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification.

Identifying functional requirements from a problem description:

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users
who might use the system and then try to identify the requirements from each user's perspective.
Here we list all functions {fi} that the system performs. Each function fi is considered as a transformation of a set of input data to some corresponding output data.

Example:-

Consider the case of the library system, where -

F1: Search Book function (fig. 3.3)

Input: an author's name

Output: details of the author's books and the location of these books in the library

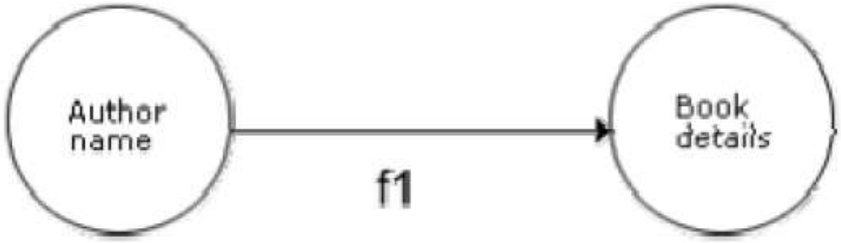


Fig: Book Function

So the function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

Documenting functional requirements:

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Example: - Withdraw Cash from ATM

R1: withdraw cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R1.1 select withdraw amount option

Input: "withdraw amount" option

Output: user prompted to enter the account type

R1.2: select account type

Input: user option

Output: prompt to enter amount

R1.3: get required amount

Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed.

2.3. NON-FUNCTIONAL REQUIREMENTS:

- Non-functional requirements are requirements that are not directly concerned with the specific services delivered by the system to its users.
- They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.
- Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole.
- Non-functional requirements are often more critical than individual functional requirements.

- System users can usually find ways to work around a system function that doesn't really meet their needs.
- However, failing to meet a non-functional requirement can mean that the whole system is unusable.

Example:

- If an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.
- Although it is often possible to identify which system components implement specific functional requirements, it is often more difficult to relate components to non-functional requirements.

Reasons are:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components.

2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required.

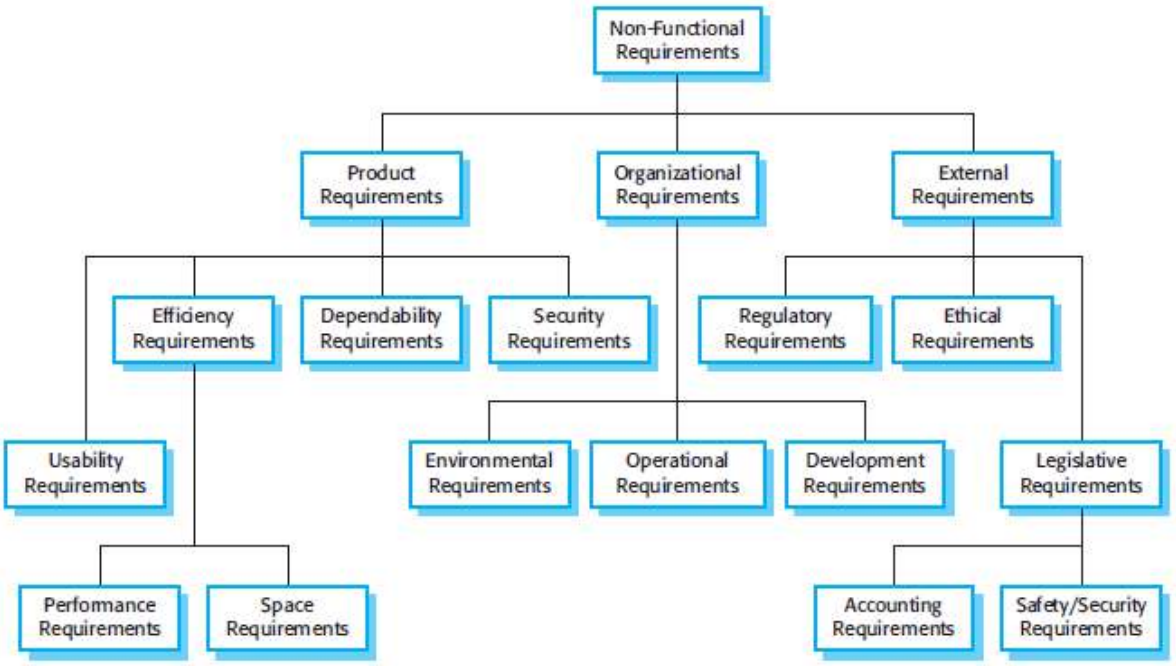


Figure. Types of non-functional requirement

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation.

Classifications of non-functional requirements are

1. **Product requirements:**
   - These requirements specify or constrain the behavior of the software.
   - Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.
2. **Organizational requirements**
   - These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization.
   - Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language,

the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.

3. **External requirements:**
    - This broad heading covers all requirements that are derived from factors external to the system and its development process.
    - Regulatory requirements set out what must be done for the system to be approved for use by a regulator, such as a central bank;
    - Legislative requirements that must be followed to ensure that the system operates within the law;
    - Ethical requirements that ensure that the system will be acceptable to its users and the general public.

Metrics for specifying non-functional requirements

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

Identifying non-functional requirements:

Nonfunctional requirements are the characteristics of the system which can not be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Nonfunctional requirements may include:

reliability issues,
performance issues,
human - computer interface issues,
interface with other external systems,
security and maintainability of the system, etc.

2.4. DOMAIN REQUIREMENTS:

- Domain requirements are derived from the application domain of the system rather than from the specific needs of system users.
- They usually include specialised domain terminology or reference to domain concepts. They may be new functional requirements in their own right, constrain existing functional requirements or set out how particular computations must be carried out.
- Because these requirements are specialised, software engineers often find it difficult to understand how they are related to other system requirements.
- Domain requirements are important because they often reflect fundamentals of the application domain. If these requirements are not satisfied, it may be impossible to make the system work satisfactorily.

2.5. USER REQUIREMENTS:

- The user requirements for a system should describe the functional and non functional requirements. So that they are understandable by system users without detailed technical knowledge.
- They should only specify the external behavior of the system and should avoid, system design characteristics.
- Consequently, if you are writing user requirements, you should not use software jargon, structured notations or formal notations, or describe the requirement by describing the system implementation.
- User requirements are written in simple language, with simple tables and forms and intuitive diagrams.

However, various problems can arise when requirements are written in natural language sentences in a text document:

1**. Lack of clarity:** It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.

2. Requirements confusion: Functional requirements, non-functional requirements, system goals and design information may not be clearly distinguished.

3. Requirements amalgamation: Several different requirements may be expressed together as a single requirement.

It is good practice to separate user requirements from more detailed system requirements in a requirements document. Otherwise, non-technical readers of the user requirements may be overwhelmed by details that are really only relevant for technicians.

Guidelines to minimize misunderstandings when writing user requirements are:

1. Invent a standard format and ensure that all requirement definitions adhere to that format.
2. Use language consistently.
3. Use text highlighting (bold, italic or colour) to pick out key parts of the requirement.
4. Avoid the use of computer jargon.

2.6. SYSTEM REQUIREMENTS:

- 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 consistent specification of the whole system.
- The system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented.

- However, at the level of detail required to completely specify a complex software system, it is impossible to exclude all design information.

There are several reasons for this:

1. There may be need to Design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different sub-systems that make up the system.
2. In most cases, systems must interoperate with other existing systems. These constrain the design, and these constraints impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified be used.

- Natural language is often used to write system requirements specifications as well as user requirements.

- **However, because system requirements are more detailed than user requirements, natural language specifications can be confusing and hard to understand:**

1. Natural language understanding relies on the specification readers and writers using the same words for the same concept.
2. A natural language requirements specification is over flexible.
3. There is no easy way to modularize natural language requirements.

- Because of these problems, requirements specifications written in natural language are prone to misunderstandings. These are often not discovered until later phases of the software process and may then be very expensive to resolve.

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.
Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was SADT (Ross, 1977) (Schoman and Ross, 1977). Now, use-case descriptions (Jacobsen, et al., 1993) and sequence diagrams are commonly used (Stevens and Pooley, 1999).
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

Figure .Notations for requirements specification

Structured language specifications:

- Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way.

- The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language but ensures that some degree of uniformity is imposed on the specification.
- Structured language notations limit the terminology that can be used and use templates to specify system requirements. They may incorporate control constructs derived from programming languages and graphical highlighting to partition the specification.
- To use a form-based approach to specifying system requirements, define one or more standard forms or templates to express the requirements.
- The specification may be structured around the objects manipulated by the system, the functions performed by the system or the events processed by the system.
- The insulin pump bases its computations of the user's insulin requirement on the rate of change of blood sugar levels. These rates of change computed using the current and previous readings.

Insulin Pump/Control Software/SRS/3.3.2	
Function	Compute insulin dose: Safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1)
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered
Destination	Main control loop
Action: CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.	
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2
Side effects	None

Figure. System requirements specification using a standard form

- When a standard form is used for specifying functional requirements, the following information should be included:

1. Description of the function or entity being specified
2. Description of its inputs and where these come from
3. Description of its outputs and where these go to
4. Indication of what other entities are used (the requires part)
5. Description of the action to be taken
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called
7. Description of the side effects (if any) of the operation.

- Using formatted specifications removes some of the problems of natural language specification. Variability in the specification is reduced and requirements are organised more effectively. However, it is difficult to write requirements in an unambiguous way, particularly when complex computations are required.

- To address this problem, you can add extra information to natural language requirements using tables or graphical models of the system.
- These can show how computations proceed, how the system state changes, how users interact with the system and how sequences of actions are performed.
- Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these.

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing. ((r2 – r1) > (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

Figure. Tabular specification of computation

Graphical models:

- Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions.
- Figure illustrates the sequence of actions when a user wishes to withdraw cash from an automated teller machine (ATM).

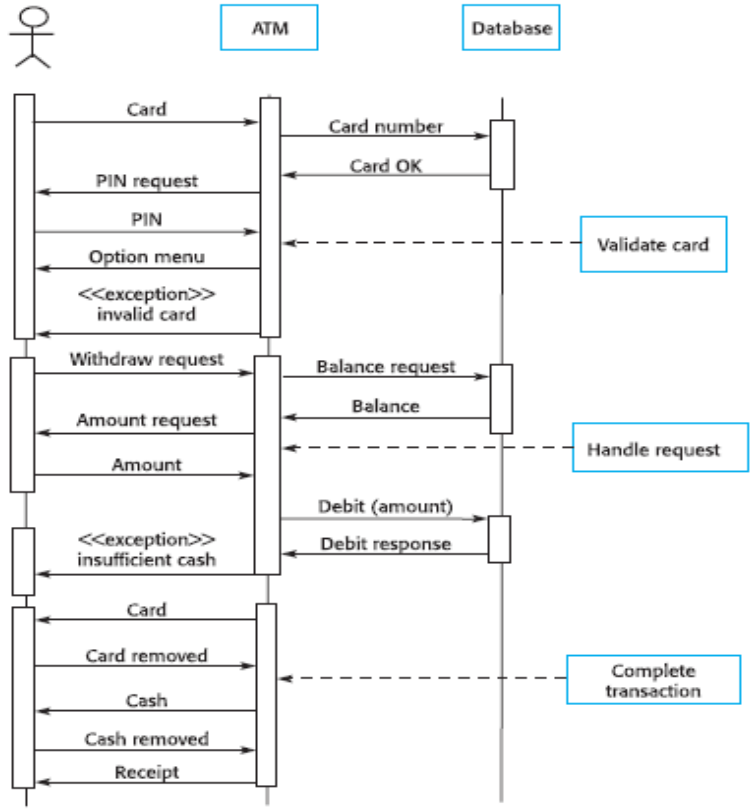


Figure. Sequence diagram of ATM withdrawal

There are three basic sub-sequences:

1. Validate card the user's card is validated by checking the card number and user's PIN.

2. Handle request the user's request is handled by the system. For a withdrawal, the database must be queried to check the user's balance and to debit the amount withdrawn. Notice the exception here if the requestor does not have enough money in their account.
3. Complete transaction The user's card is returned and, when it is removed, the cash and receipt are delivered.

2.7. Software document (or)
Software requirements document (or)
Software Rrequirements Specification (SRS)

- The software requirements document (or) SRS is an official statement of what the system developers should implement.
- A *software requirements specification* (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written.
- In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.
- It should include both the user requirements for a system and a detailed specification of the system requirements.
- Sometimes, the user and system requirements are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document.
- The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software.

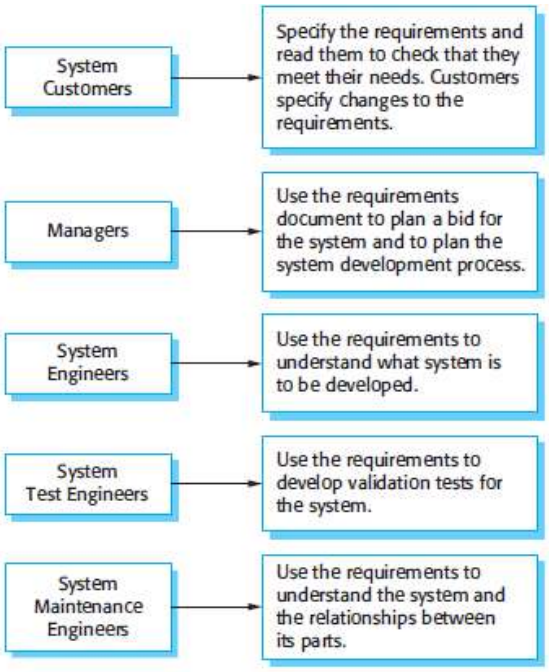


Figure. Users of a requirements document

- IEEE (Institute of Electrical and Electronics Engineers) standard suggest the following structure for requirements documents.

- Although IEEE standard is not ideal, it contains good deal of good advice on how to write requirements and how to avoid problems.

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

Figure. The structure of a requirements document

Software Requirements Specification Template

Table of Contents

1. Introduction

1.1 Purpose

1.2 Document Conventions

1.3 Intended Audience and Reading Suggestions

1.4 Project Scope

1.5 References

2. Overall Description

2.1 Product Perspective

2.2 Product Features

2.3 User Classes and Characteristics

2.4 Operating Environment

2.5 Design and Implementation Constraints
2.6 User Documentation
2.7 Assumptions and Dependencies
3. System Features
3.1 System Feature 1
3.2 System Feature 2 (and so on)
4. External Interface Requirements
4.1 User Interfaces
4.2 Hardware Interfaces
4.3 Software Interfaces
4.4 Communications Interfaces
5. Other Nonfunctional Requirements
5.1 Performance Requirements
5.2 Safety Requirements
5.3 Security Requirements
5.4 Software Quality Attributes
6. Other Requirements
Appendix A: Glossary
Appendix B: Analysis Models

Properties of a good SRS document:

• The important properties of a good SRS document are the following:

 Concise. The SRS document should be concise and at the same time unambiguous, consistent, and complete.

 Structured. It should be well-structured. A well-structured document is easy to understand and modify.

 Black-box view. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system.

 Conceptual integrity. It should show conceptual integrity so that the reader can easily understand it.

 Response to undesired events. It should characterize acceptable responses to undesired events.

Problems without a SRS document

• The important problems that an organization would face if it does not develop an SRS document are as follows:

 Without developing the SRS document, the system would not be implemented according to customer needs.

 Software developers would not know whether what they are developing is what exactly required by the customer.

 Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.

 It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

2.8. REQUIREMENTS ENGINEERING PROCESSES:

- ❖ Requirements engineering (RE) refers to the process of **defining, documenting and maintaining requirements.**
- ❖ Requirements engineering emphasizes the use of systematic and repeatable techniques that ensure the **completeness, consistency, and relevance of the system requirements.**
- ❖ The goal of the requirements engineering process is to create and maintain a system requirements document.

Requirements engineering process includes **four sub-processes.**

1) **Feasibility study:** Assessing whether the system is useful to the business.
2) **Elicitation and analysis** :
   - **Requirements elicitation** is the process of *discovering, reviewing, documenting, and understanding* the user's needs and constraints for the system.
   - **Requirements analysis** is the process of *refining* the user's needs and constraints.
3) **Specification:** Converting these requirements into some standard form. It is the process of documenting the user's needs and constraints clearly and precisely.
4) **Validation: C**hecking that the requirements actually define the system that the customer wants.

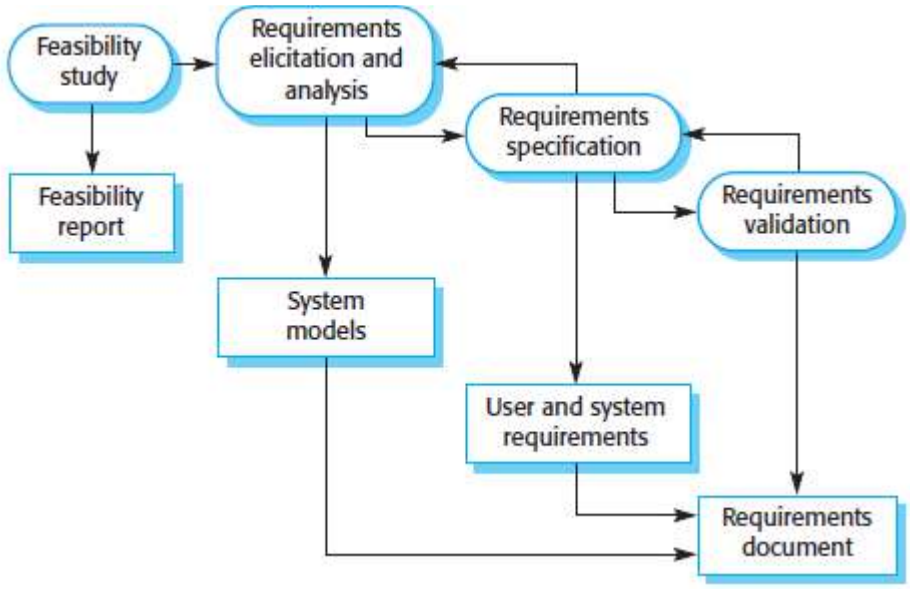


Figure. The requirements engineering process

- Figure illustrates the relationship between the activities. It also shows the documents produced at each stage of the requirements engineering process.
- The activities are concerned with the discovery, documentation and checking of requirements.
- In all systems, normally requirements change frequently.
  - Reasons for changing requirements:
  - The people involved, develop a *better understanding* of what they want the software to do;
  - The organisation buying the *system changes*;
  - Modifications are made to the system's *hardware, software and organisational environment.*
- The process of managing these changing requirements is called requirements management.

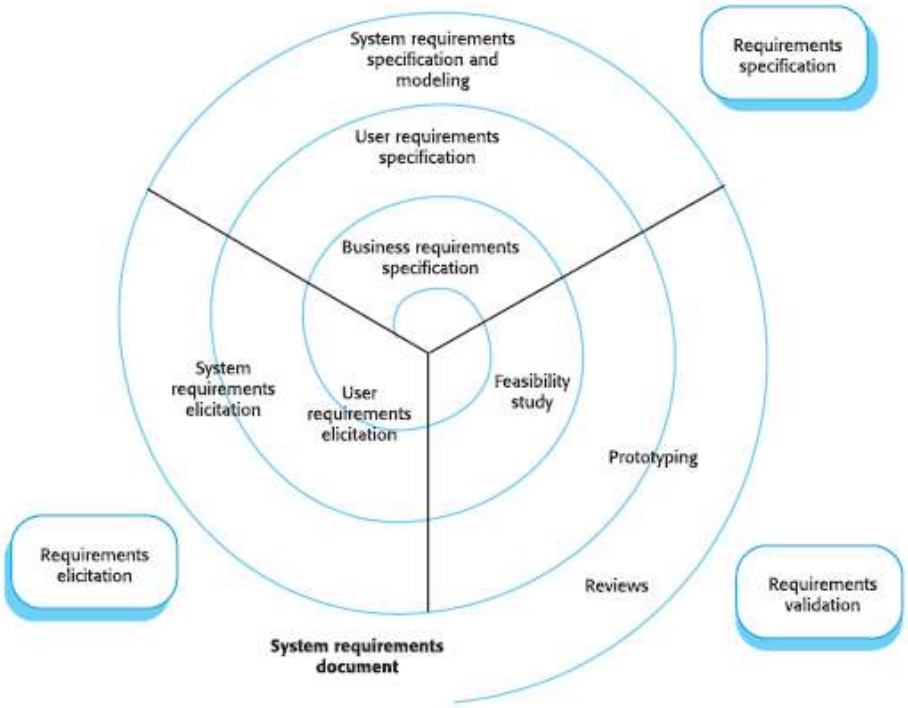


Figure .Spiral models of requirements engineering processes

Spiral model of requirements engineering process

- An alternative perspective on the requirements engineering process is spiral model of requirements engineering process. This presents the process as a three-stage activity where the activities are organized as an iterative process around a spiral.
- The amount of time and effort devoted to each activity in iteration depends on the stage of the overall process and the type of system being developed.
- Early in the process, most effort will be spent on understanding *high-level business and non-functional requirements and the user requirements*.
- Later in the process, in the outer rings of the spiral, more effort will be devoted to *system requirements engineering and system modeling.*
- This spiral model accommodates approaches to development in which the requirements are developed to different levels of detail. The number of iterations around the spiral can vary, so the spiral can be exited after some or all of the user requirements have been elicited.
- If the prototyping activity shown under requirements validation is extended to include iterative development, this model allows the requirements and the system implementation to be developed together.

2.9. FEASIBILITY STUDIES:

For all new systems, the requirements engineering process should start with a feasibility study. **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.

A feasibility study is a short, focused study that aims to answer a number of questions:

1. Does the system contribute to the overall objectives of the organisation?
2. Can the system be implemented using current technology and within given cost and schedule constraints?
3. Can the system be integrated with other systems which are already in place?

Carrying out a feasibility study involves 3 activities

1) Information assessment
2) Information collection
3) Report writing.

1) **Information assessment:**
   - The information assessment phase identifies the information that is required to answer the three questions set out above.
   - Once the information have been identified, then talk with information sources to discover the answers to these questions.

Some examples of possible questions that may be put are:

a) How would the organization cope if this system were not implemented?
b) What are the problems with current processes and how would a new system help alleviate these problems?
c) What direct contribution will the system make to the business objectives and requirements?
d) Can information be transferred to and from other organizational systems?
e) Does the system require technology that has not previously been used in the organization?
f) What must be supported by the system and what need not be supported?

2) Information collection :
   - Consult with information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts and end-users of the system.
   - Feasibility study should be completed in two or three weeks.

3) Report writing:
   - Once information is collected, write the feasibility study report. Report can contain a *recommendation* about whether or not the system development should continue.
   - Report can propose ***changes*** *to the scope, budget and schedule of the system and* ***suggest*** *further high-level requirements for the system*.

2.10. REQUIREMENTS ELICITATION AND ANALYSIS:

- Software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.
- Requirements elicitation and analysis may involve a variety of people in an organisation.
- **The term stakeholder is used to refer to any person or group who will be affected by the system, directly or indirectly**.
    - Stakeholders include end-users who interact with the system and everyone else in an organization that may be affected by its installation.
    - Other system stakeholders may be engineers who are developing or maintaining related systems, business managers, domain experts and trade union representatives.

Eliciting and understanding stakeholder requirements is difficult for several reasons:

1) Stakeholders often ***don't know what they want*** from the computer system except in the most general terms.
2) Stakeholders naturally express requirements in their own terms and with ***implicit knowledge*** of their own work.
3) *Different* ***stakeholders have different requirements***.
4) ***Political factors*** may influence the requirements of the system.

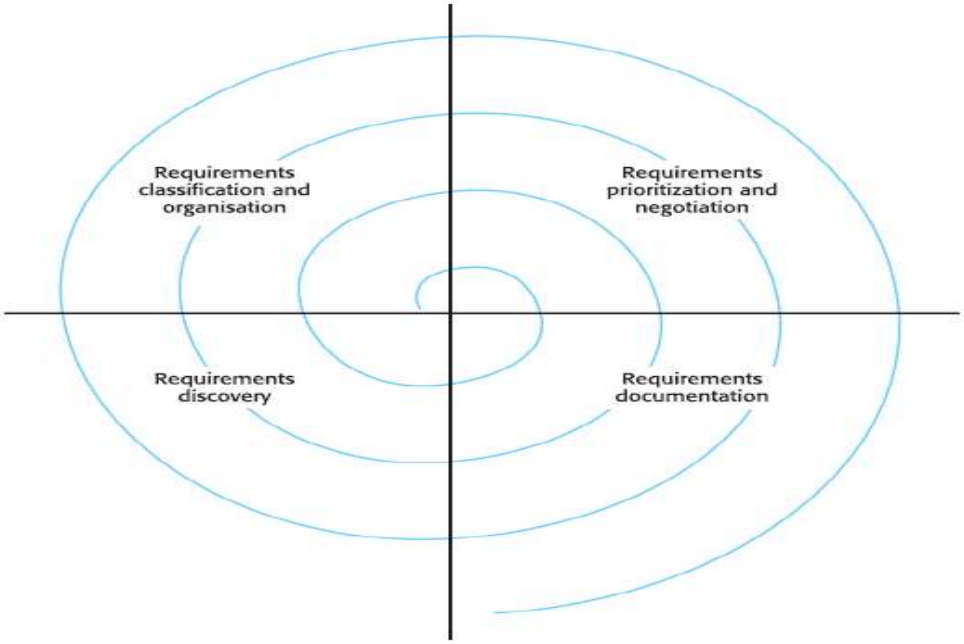


Figure. The requirements elicitation and analysis process

- The activities are interleaved as the process proceeds from the inner to the outer rings of the spiral.

The process activities are:

1. **Requirements discovery:**
   This is the process of interacting with stakeholders in the system to collect their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
2. **Requirements classification and organization :**
   This activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.

3. **Requirements prioritization and negotiation :**
   Inevitably, where multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements, and finding and resolving requirements conflicts through negotiation.
4. **Requirements documentation:**
   The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

2.10.1 Requirements discovery:

Requirements discovery is the process of gathering information about the proposed 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.

Interact with stakeholders through interviews and observation, and may use scenarios and prototypes to help with the requirements discovery.

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

For example: 1. System stakeholders for a bank ATM include:

a) Current bank customers who receive services from the system
b) Representatives from other banks who have reciprocal agreements that allow each other's ATMs to be used
c) Managers of bank branches who obtain management information from the system
d) Counter staff at bank branches who are involved in the day-to-day running of the system
e) Database administrators who are responsible for integrating the system with the bank's customer database
f) Bank security managers who must ensure that the system will not pose a security hazard
g) The bank's marketing department who are likely be interested in using the system as a means of marketing the bank
h) Hardware and software maintenance engineers who are responsible for maintaining and upgrading the hardware and software
i) National banking regulators who are responsible for ensuring that the system conforms to banking regulations

For example: 2. System stakeholders for the mental healthcare patient information system include:

1. Patients whose information is recorded in the system.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Healthcare managers who obtain management information from the system.
8. 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.

In addition to system stakeholders, requirements may come from the application domain and from other systems that interact with the system being specified. All of these must be considered during the requirements elicitation process.

Techniques used for requirements discovery are

1) Viewpoint
2) Interviewing
3) Scenarios
4) Ethnography

1) Viewpoints:

- The requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints, where each viewpoint presents a sub-set of the requirements for the system.
- Each viewpoint provides a fresh perspective on the system, but these perspectives are not completely independent—they usually overlap so that they have common requirements.
- A key strength of viewpoint-oriented analysis is that it recognizes multiple perspectives and provides a framework for discovering conflicts in the requirements proposed by different stakeholders.
- Viewpoints can be used as a way of classifying stakeholders and other sources of requirements.

Three generic types of viewpoint are

a) **Interactor viewpoints:** It represents people or other systems that interact directly with the system. In the bank ATM system, examples of interactor viewpoints are the bank's customers and the bank's account database.

b) **Indirect viewpoints:** It represents stakeholders who do not use the system themselves but who influence the requirements in some way. In the bank ATM system, examples of indirect viewpoints are the management of the bank and the bank security staff.

c) **Domain viewpoints**: It represents domain characteristics and constraints that influence the system requirements. In the bank ATM system, an example of a domain viewpoint would be the standards that have been developed for interbank communications.

- Interactor viewpoints provide detailed system requirements covering the system features and interfaces.
- Indirect viewpoints are more likely to provide higher-level organizational requirements and constraints.
- Domain viewpoints normally provide domain constraints that apply to the system.

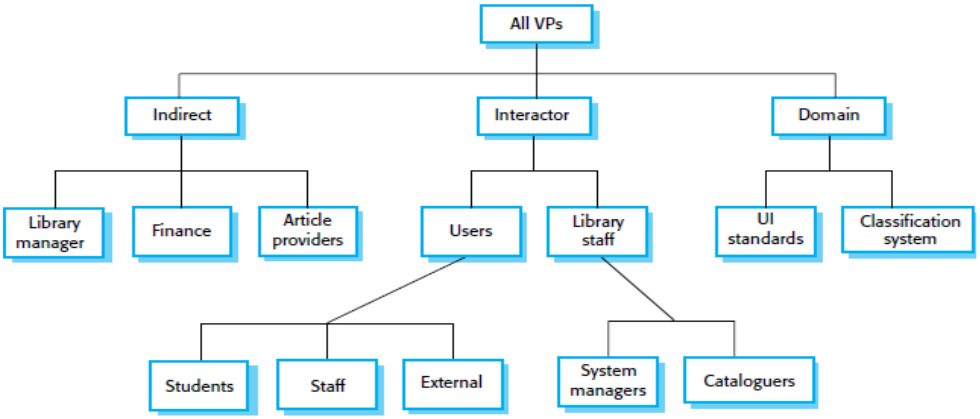


Figure.Viewpoints in LIBSYS

Engineering viewpoints may be important for two reasons.

a) The engineers developing the system may have experience with similar systems and may be able to suggest requirements from that experience.

b) Technical staff who have to manage and maintain the system may have requirements that will help simplify system support.

- Viewpoints that provide requirements may come from the marketing and external affairs departments in an organisation. This is especially true for web-based systems, particularly e-commerce systems and shrink-wrapped software products.
- Web-based systems must present a favourable image of the organisation as well as deliver functionality to the user. For software products, the marketing department should know what system features will make the system more marketable to potential buyers.
- Viewpoints in the same branch are likely to share common requirements.
- Once viewpoints have been identified and structured, try to identify the most important viewpoints and start with them when discovering system requirements.

2) Interviewing:

- Formal or informal interviews with system stakeholders are part of most requirements engineering processes.
- In these interviews, the requirements engineering team puts questions to stakeholders about the system that they use and the system to be developed. Requirements are derived from the answers to these questions.

Interviews may be of two types:

(1) **Closed interviews** where the stakeholder answers a predefined set of questions.
(2) **Open interviews** where there is no predefined agenda.

- **Interviews are good for getting an overall understanding of what stakeholders do**, how they might interact with the system and the difficulties that they face with current systems.
- **People like talking about their work and are usually happy to get involved in interviews.** However, interviews are not so good for understanding the requirements from the application domain.

It is hard to elicit domain knowledge during interviews for two reasons:

(1) All *application specialists use terminology and jargon that is specific to a domain*.
(2) 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.

Two characteristics of Effective interviewers:

(1) They are **open-minded, avoid preconceived ideas** about the requirements and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, they are willing to change their mind about the system.
(2) They **prompt the interviewee to start discussions with a question**, a requirements proposal or by suggesting working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information. Most people find it much easier to talk in a defined context rather than in general terms.

Interviews should be used alongside other requirements elicitation techniques.

3) Scenarios:

- Scenarios can be particularly useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions.
- *Each scenario covers one or more possible interactions*. Several forms of scenarios have been developed, each of which provides different types of information at different levels of detail about the system.
- The scenario starts with an outline of the interaction, and, during elicitation, details are added to create a complete description of that interaction.

A scenario may include:

1. A description of what the system and users expect when the **scenario starts**
2. A description of the **normal flow of events** in the scenario
3. A description of **what can go wrong and how this is handled**
4. Information about other **activities that might be going on at the same time**
5. A description of the **system state when the scenario finishes**.

- Scenario-based elicitation can be carried out informally, where the requirements engineer works with stakeholders to identify scenarios and to capture details of these scenarios.
- Scenarios may be written as text, supplemented by diagrams, screen shots, and so on.
- Alternatively, a more structured approach such as event scenarios or usecases may be adopted. As an example of a simple text scenario, consider how a user of the LIBSYS library system may use the system.

Initial assumption: The user has logged on to the LIBSYS system and has located the journal containing the copy of the article.

Normal: The user selects the article to be copied. The system prompts the user to provide subscriber information for the journal or to indicate a method of payment for the article. Payment can be made by credit card or by quoting an organisational account number.

The user is then asked to fill in a copyright form that maintains details of the transaction and submit it to the LIBSYS system.

The copyright form is checked and, if it is approved, the PDF version of the article is downloaded to the LIBSYS working area on the user's computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed. If the article has been flagged as 'print-only' it is deleted from the user's system once the user has confirmed that printing is complete.

What can go wrong: The user may fail to fill in the copyright form correctly. In this case, the form should be re-presented to the user for correction. If the resubmitted form is still incorrect, then the user's request for the article is rejected.

The payment may be rejected by the system, in which case the user's request for the article is rejected.

The article download may fail, causing the system to retry until successful or the user terminates the session.

It may not be possible to print the article. If the article is not flagged as 'print-only' it is held in the LIBSYS workspace. Otherwise, the article is deleted and the user's account credited with the cost of the article.

Other activities: Simultaneous downloads of other articles.

System state on completion: User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.

Figure. Scenario for article downloading in LIBSYS

Use-cases:

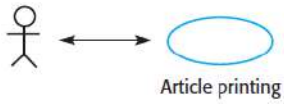


Figure . A simple use-case for article printing

- Use-cases are a scenario-based technique for requirements elicitation which were first introduced in the Objectory method. They have now become a fundamental feature of the UML notation for describing object-oriented system models. A use-case identifies the type of interaction and the actors involved.

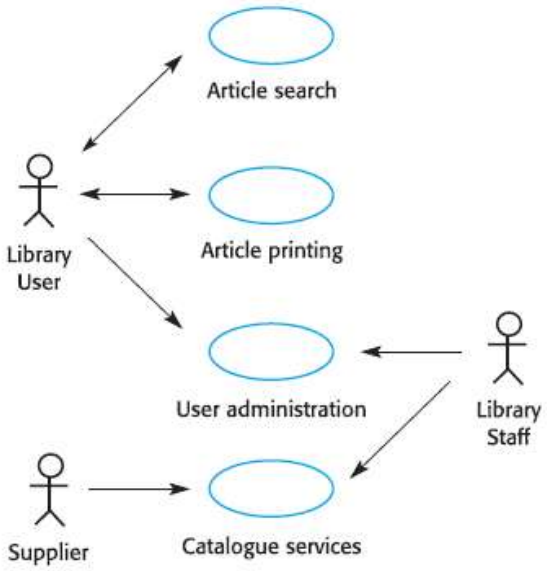


Figure . Use cases for the library system

- *Actors in the process are represented as stick figures*, and each class of interaction is represented as a named ellipse.
- The set of *use-cases represents all of the possible interactions* to be represented in the system requirements.
- *Use-cases identify the individual interactions with the system*. They can be documented with text or linked to UML (Unified Modelling Language) models that develop the scenario in more detail.

Sequence diagrams:

- Sequence diagrams are often used to add information to a use-case. These sequence diagrams show the actors involved in the interaction, the objects they interact with and the operations associated with these objects.

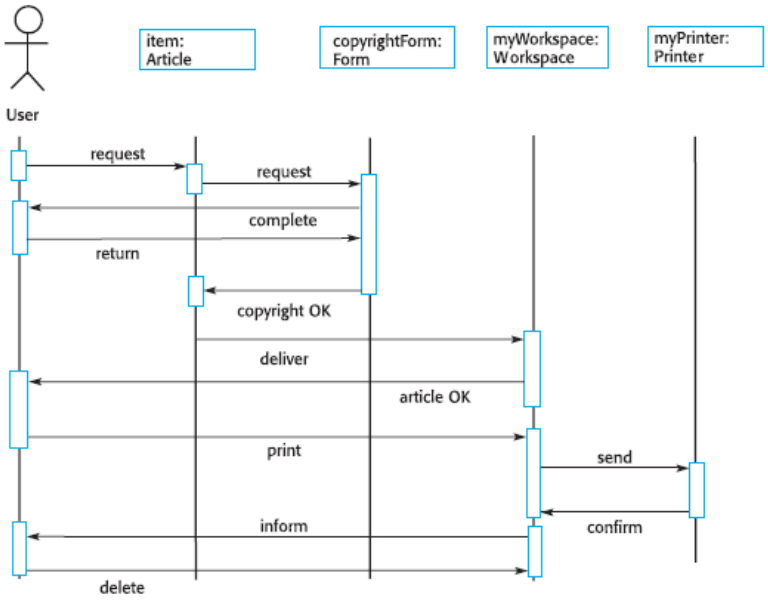


Figure. System interactions for article printing

- In Figure, there are four objects of classes—Article, Form, Workspace and Printer—involved in this interaction. The sequence of actions is from top to bottom, and the labels on the arrows between the actors and objects indicate the names of operations.
- Essentially, a user request for an article triggers a request for a copyright form. Once the user has completed the form, the article is downloaded and sent to the printer. Once printing is complete, the article is deleted from the LIBSYS workspace.
- Scenarios and use-cases are effective techniques for eliciting requirements for interactor viewpoints, where each type of interaction can be represented as a usecase.
- They can also be used in conjunction with some indirect viewpoints where these viewpoints receive some results from the system.

Drawbacks:

- They are not as effective for eliciting constraints or high-level business and non-functional requirements from indirect viewpoints or for discovering domain requirements.

4) **Ethnography:**

Ethnography is an observational technique that can be used to understand social and organizational requirements.

- An analyst **immerses him or herself in the working environment** where the system will be used. He or she observes the day-to-day work and notes made of the actual tasks in which participants are involved.
- The value of ethnography is that it helps analysts **discover implicit system requirements** that reflect the actual rather than the formal processes in which people are involved.
- **Social and organizational factors that affect the work** but that are not obvious to individuals may only become clear when noticed by an unbiased observer.

Ethnography is particularly effective at discovering two types of requirements:

1. 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.
2. Requirements that are derived from cooperation and awareness of other people's activities.

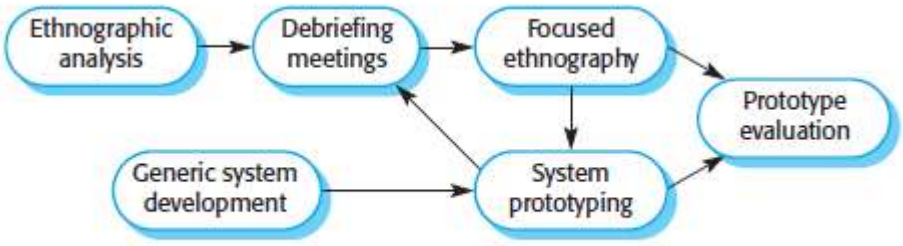


Figure. Ethnography and prototyping for requirements

- Ethnography may 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.
- Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques.

Drawbacks:

(1) This approach is not appropriate for discovering organisational or domain requirements.
(2) Ethnographic studies cannot always identify new features that should be added to a system.
(3) Ethnography is not a complete approach to elicitation on its own, and it should be used to complement other approaches, such as use-case analysis.

2.11. REQUIREMENTS VALIDATION

- Requirements validation is concerned with showing that the requirements actually define the system that the customer wants. Requirements validation overlaps analysis in that it is concerned with finding problems with the requirements.
- Requirements validation is important because errors in a requirements document can lead to extensive rework costs when they are discovered during development or after the system is in service. The cost of fixing a requirements problem by making a system change is 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 and then the system must be tested again.
- During the requirements validation process, checks should be carried out on the requirements in the requirements document.

These checks include:

1. **Validity checks** :A user may think that a system is needed to perform certain functions.
2. **Consistency checks**: Requirements in the document should not conflict.
3. **Completeness checks**: The requirements document should include requirements, which define all functions, and constraints intended by the system user.
4. **Realism checks**: Using knowledge of existing technology, the requirements should be checked to ensure that they could actually be implemented.
5. **Verifiability:** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable.

A number of requirements validation techniques can be used in conjunction or individually:

i) **Requirements reviews:** The requirements are analysed systematically by a team of reviewers.

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

iii) **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.

1. Requirements reviews:

- A *requirements review is a manual process* that involves people from both client and contractor organisations.
- They check the requirements document for anomalies and omissions. The review process may be managed in the same way as program inspections.
- Alternatively, it may be organised as a broader activity with different people checking different parts of the document.
- Requirements reviews can be informal or formal.
- **Informal reviews** simply involve contractors discussing requirements with as many system stakeholders as possible. Many problems can be detected simply by talking about the system to stakeholders before making a commitment to a formal review.
- In a **formal requirements review**, the development team should '*walk' the client through the system requirements,* explaining the implications of each requirement.
- The review team should check each requirement for consistency as well as check the requirements as a whole for completeness.

Reviewers may also check for:

1. **Verifiability:** Is the requirement as stated realistically testable?
2. **Comprehensibility**: Do the procurers or end-users of the system properly understand the requirement?
3. **Traceability:** Is the origin of the requirement clearly stated? You may have to go back to the source of the requirement to assess the impact of a change. Traceability is important as it allows the impact of change on the rest of the system to be assessed.
4. **Adaptability:** Is the requirement adaptable? That is, can the requirement be changed without large-scale effects on other system requirements?

- **Conflicts, contradictions, errors and omissions in the requirements should be pointed out by reviewers and formally recorded in the review report.** It is then up to the users, the system procurer and the system developer to negotiate a solution to these identified problems.

2.12. REQUIREMENTS MANAGEMENT

- The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems.
- Because the problem cannot be fully defined, the software requirements are bound to be incomplete.
- During the software process, the **stakeholders' understanding** of the problem is constantly changing. These requirements must then evolve to **reflect this changed problem view**.
- Furthermore, once a system has been installed**, new requirements** inevitably emerge.
- It is hard for users and system customers to anticipate what effects the new system will have on the organisation.

Once end-users have experience of a system, they discover new needs and priorities:

1. Large systems usually have a diverse user community where users have different requirements and priorities. These may be conflicting or contradictory.
2. The people who pay for a system and the users of a system are rarely the same people.
3. The business and technical environment of the system changes after installation, and these changes must be reflected in the system.

- Requirements management is the process of understanding and controlling changes to system requirements.
- You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.
- You need to establish a formal process for making change proposals and linking these to system requirements.
- The process of requirements management should start as soon as a draft version of the requirements document is available, but you should start planning how to manage changing requirements during the requirements elicitation process.

Enduring and volatile requirements:

- Requirements evolution during the RE process and after a system has gone into service is inevitable.
- Developing software requirements focuses attention on software capabilities, business objectives and other business systems.
- As the requirements definition is developed, you normally develop a better understanding of users' needs.
- This feeds information back to the user, who may then propose a change to the requirements. Furthermore, it may take several years to specify and develop a large system. Over that time, the system's environment and the business objectives change, and the requirements evolve to reflect this.

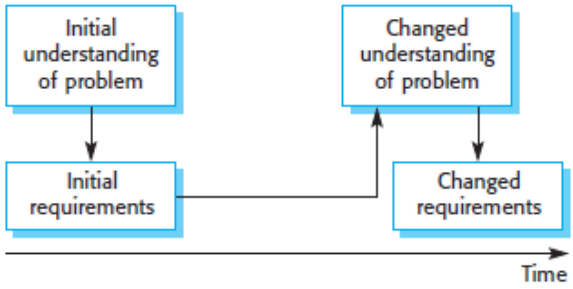


Figure. Requirements evolution

From an evolution perspective, requirements fall into two classes:

1. **Enduring requirements:** These are relatively stable requirements that **derive from the core activity of the organisation** and which relate directly to the domain of the system. For example, in a hospital, there will always be requirements concerned with patients, doctors, nurses and treatments.
2. **Volatile requirements:** These are requirements that are **likely to change during the system development process** or after the system has been become operational. An example would be requirements resulting from government healthcare policies.

1) Requirements management planning:

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

Requirements management is very expensive. For each project, the planning stage establishes the level of requirements management detail that is required.

Requirement Type	Description
Mutable requirements	Requirements which change because of changes to the environment in which the organisation is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.
Emergent requirements	Requirements which emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements.
Consequential requirements	Requirements which result from the introduction of the computer system. Introducing the computer system may change the organisation's processes and open up new ways of working which generate new system requirements.
Compatibility requirements	Requirements which depend on the particular systems or business processes within an organisation. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.

Figure. Classification of volatile requirements

During the requirements management stage, you have to decide on:

1. **Requirements identification:**
   Each requirement must be uniquely identified so that it can be cross-referenced by other requirements and so that it may be used in traceability assessments.
2. **A change management process:**
   This is the set of activities that assess the impact and cost of changes. .
3. **Traceability policies:**
   These policies define the *relationships between requirements*, and between the requirements and the system design that should be recorded and how these records should be maintained.
4. **CASE 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.

There are three types of traceability information that may be maintained:

1. **Source traceability** information links the **requirements to the stakeholders** who proposed the requirements and to the rationale for these requirements. When a change is proposed, you use this information to find and consult the stakeholders about the change.
2. **Requirements traceability** information **links dependent requirements** within the requirements document. Use this information to assess how many requirements are likely to be affected by a proposed change and the extent of consequential requirements changes that may be necessary.
3. **Design traceability** information **links the requirements to the design modules** where these requirements are implemented. Use this information to assess the impact of proposed requirements changes on the system design and implementation.

- Traceability information is often represented using traceability matrices, which relate requirements to stakeholders, each other or design modules.
- In a requirements traceability matrix, each requirement is entered in a row and in a column in the matrix. Where dependencies between different requirements exist, these are recorded in the cell at the row/column intersection.

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			R		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

Figure .A traceability matrix

- A 'D' in the row/column intersection illustrates that the requirement in the row *depends on the requirement named in the column*; an *'R' means that there is some other, weaker relationship between the requirements*.
- Traceability matrices may be used when a small number of requirements have to be managed, but they become unwieldy and expensive to maintain for large systems with many requirements.
- For large systems, capture traceability information in a requirements database where each requirement is explicitly linked to related requirements.
- Assess the impact of changes by using the database browsing facilities. Traceability matrices can be generated automatically from the database. Requirements management needs automated support; the CASE tools for this should be chosen during the planning phase.

Tools are needed to support:

1. **Requirements storage:**
   The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
2. **Change management:**
   The process of change management is simplified if active tool support is available.
3. **Traceability management:**
   Tool support for traceability allows related requirements to be discovered. Some tools use natural language processing techniques to help you discover possible relationships between the requirements.

2) Requirements change management:

Requirements change management should be applied to all proposed changes to the requirements. The advantage of using a formal process for change management is that all change proposals are treated consistently and that changes to the requirements document are made in a controlled way.

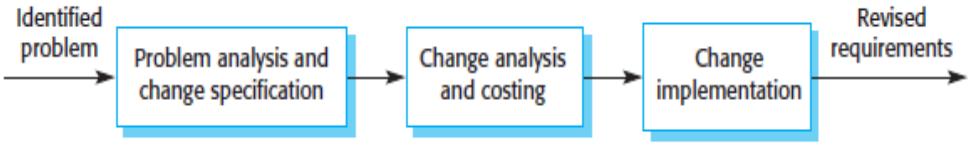


Figure. Requirements change management

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. The results of the analysis are fed back to the change requestor, and sometimes a more specific requirements change proposal is then made.
2. **Change analysis and costing**:
   - ✓ The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements.
   - ✓ The cost of making the change is estimated in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether to proceed with the requirements change.
3. **Change implementation:**
   - ✓ The requirements document and, where necessary, the system design and implementation are modified.
   - ✓ Organize the requirements document so that you can make changes to it without extensive rewriting or reorganization.
   - ✓ 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.

2.13. CLASSICAL ANALYSIS

Specification document must satisfy two mutually contradictory requirements

1. It must be **clear and intelligible to client**
   - ✓ Client probably not a computer expert
   - ✓ Client must understand it in order to authorize
2. It must be **complete and detailed** - Sole source of information available to the design team

Classification of Requirement specification techniques

(1) **Informal**: Written in a natural language
(2) **Formal**: Techniques such as Petri nets and Z
(3) **Semiformal:** Techniques between informal and formal.eg. structured system analysis

2.14. STRUCTURED SYSTEMS ANALYSIS

The structured system analysis is a technique in which the system requirements are converted into specifications and then into computer programs, hardware configurations and related manual procedures.

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.

DATA FLOW DIAGRAM

- A Data Flow Diagram (DFD) is a graphical representation of the "flow" of data through an information system.
- It is common practice to draw a System Context Diagram first which shows the interaction between the system and outside entities.
- The DFD is designed to show how a system is divided into smaller portions and to highlight the flow of data between those parts.
- Level 0 DFD i.e. Context level DFD should depict the system as a single.
- Primary input and primary output should be carefully identified.
- Information flow from continuity must be maintained from level to level

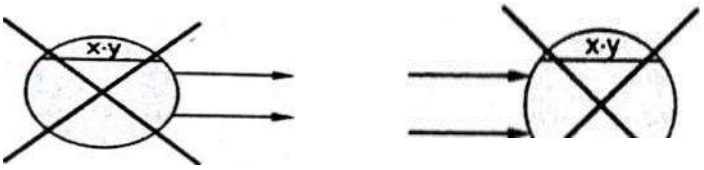
Four basic symbols

Symbol	Notation
External Entity: External entities are objects outside the system, with which the system communicsates. External entities are sources and destinations of the system's inputs and outputs.	External Entity
Process : A process tranforms incoimg data flow into outgoing data flow	Level / Process
Transition: It represents the flow of information from one entity to another	
Data Store: Data store are repositories of data in the system. They are sometimes also referred to as files or databases.	

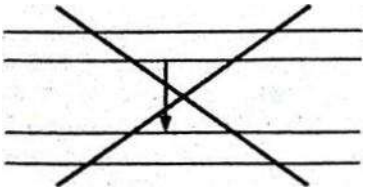
Figure .symbol of structured system analysis

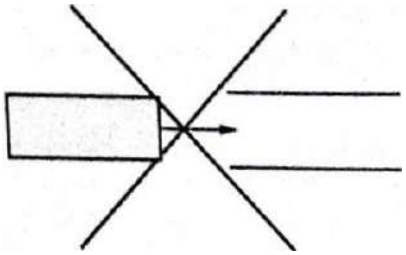
Rules for Designing DFD:

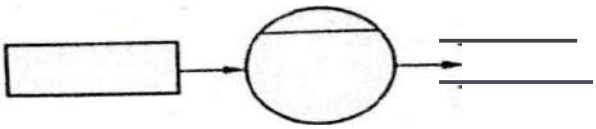
1. No process can have only outputs or only inputs. The process must have both Outputs and inputs.



2. The verb phrases in the problem description can be identified as processes in the system.
3. There should not be a direct flow between data stores and external entity. This flow should go through a process.







4. Data store labels should be noun phrases from problem description.
5. No data should move directly between external entities. The data flow should go through a process.
6. Generally source and sink labels are noun phrases.

Step 1: Draw the Data Flow Diagram (DFD)

- A pictorial representation of all aspects of the logical data flow
  - ✓ Logical data flow — What happens
  - ✓ Physical data flow — How it happens
- Any non-trivial product contains many elements
- DFD is developed by stepwise refinement
- For large products a hierarchy of DFDs instead of one DFD
- Constructed by identifying data flows: Within requirements document or rapid prototype

Step 2: Decide what sections to computerize and how (batch or online)

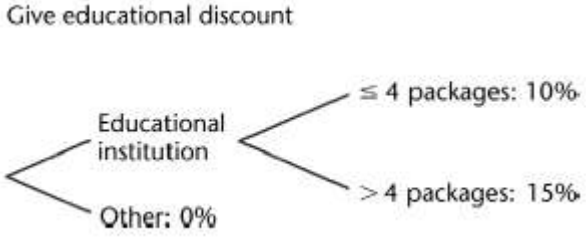
- ✓ Depending on client's needs and budget limitations ƒ
- ✓ Cost-benefit analysis is applied

Step 3: Determine details of data flows

- ✓ Decide what data items must go into various data flows
- ✓ Stepwise refinement of each flow
- ✓ For larger products, a data dictionary is generated.
- ✓ **Data dictionary -** keeps track of all data element.
  - o A data dictionary is a collection of data about data.
  - o It maintains information about the definition, structure, and use of each data element that an organization uses.

Step 4: Define logic of processes

- ✓ Determine what happens within each process
- ✓ Use of decision trees to consider all cases



Step 5: Define data stores

- ✓ Exact contents of each store and its representation (format)

Step 6: Define physical resources

- ✓ File names, organization (sequential, indexed, etc.), storage medium, and records
- ✓ If a database management system (DBMS) used: Relevant information for each table

Step 7: Determine input-output specifications

- ✓ Input forms and screens
- ✓ Printed outputs

Step 8: Determine sizing

- ✓ Computing numerical data to determine hardware requirements
- ✓ Volume of input (daily or hourly)
- ✓ Frequency of each printed report and its deadline
- ✓ Size and number of records of each type to pass between CPU and mass storage
- ✓ Size of each file
- ✓

Step 9: Determine hardware requirements

- ✓ Use of sizing information to determine mass storage requirements
- ✓ Mass storage for backup
- ✓ Determine if client's current hardware system is adequate

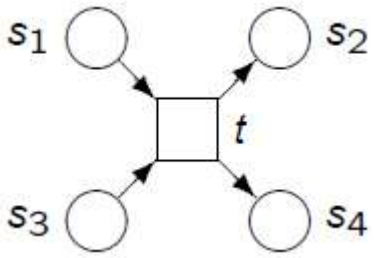
After approval by client: Specification document is handed to design team, and software process continues

2.15. PETRI NETS

- Petri nets are a basic model of parallel and distributed systems. The basic idea is to describe state changes in a system with transitions.
- Petri nets — Formal technique for describing concurrent interrelated activities
- Invented by Carl Adam Petri, 1962

Petri net Consists of four parts

(1) A set of places
(2) A set of transitions
(3) An input function
(4) An output function



- Petri nets contain places ○ (Stelle) and transitions □ or | (Transition) that may be connected by directed arcs.
- Transitions symbolise actions; places symbolise states or conditions that need to be met before an action can be carried out.

● Marking of a Petri net

- ✓ Assignment of tokens
- ✓ Tokens enable transitions

● Petri nets are non-deterministic

Petri nets and their firing rule:

A place may contain several tokens, which may be interpreted as resources.

- There may be several input and output arcs between a place and a transition.
- The number of these arcs is represented as the weight of a single arc.
- A transition is enabled if its each input place contains at least as many tokens as the corresponding input arc weight indicates.
- When an enabled transition is fired, its input arc weights are subtracted from the input place markings and its output arc weights are added to the output place markings.

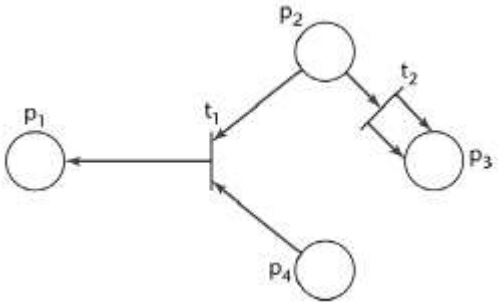


Fig.A Pertinet

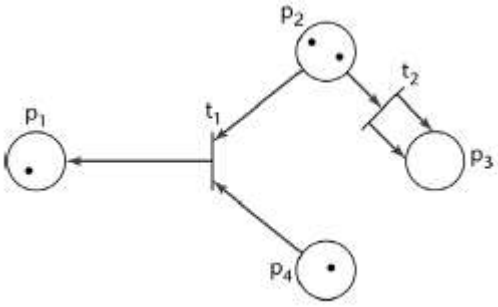


Fig. A marked Petrinet

More formally, a Petri net is a 4-tuple C = (P, T, I, O)

P = {p , p ,…,p } is a finite set of *places*, n ≥ 0

T = {t_1, t_2,…,t_m} is a finite set of *transitions*, m ≥ 0, with P and T

I : T → P^∞ is the *input* function, a mapping from transitions to bags of places

O : T → P^∞ is the *output* function, a mapping from

Petri net in the above figure has,

Set of places P is	{p_1, p_2, p_3, p_4}
Set of transitions T is	{t_1, t_2}
Input functions:	I(t_1) = {p_2, p_4}
	I(t_2)= {p_2}
Output functions:	O(t_1) = {p_1}
	O(t_2) = {p_3, p_3}

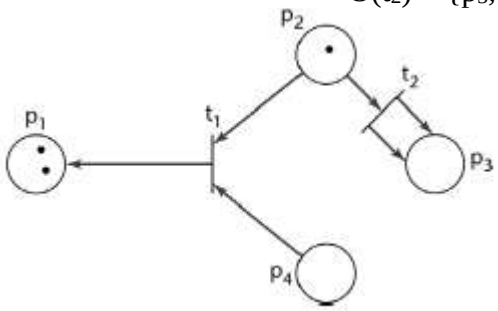


Fig. After transision t1 fires

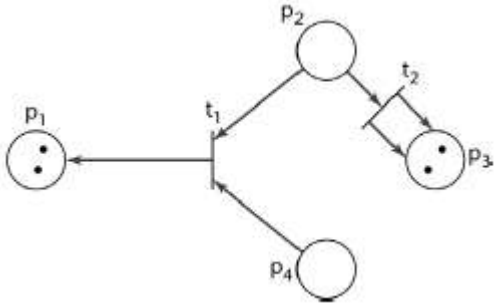


Fig. After transition t2 fire

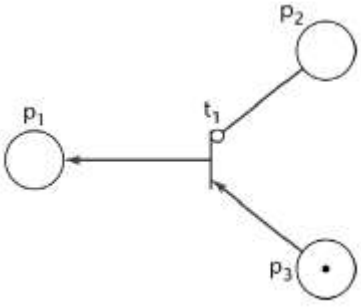


Fig. A petrinet with an inhibitor arc

Inhibitor arcs:

An inhibitor arc is marked by a small circle, not an arrowhead. Transition t_1 is enabled.

A marked Petri net is then a 5-tuple (P, T, I, O, M).

In general, a transition is enabled if there is at least one token on each (normal) input arc, and **no tokens on any inhibitor input arcs.**

CASE Tools for Classical Analysis

● Two classes of CASE tools are helpful during classical analysis

● A graphical tool for drawing data flow diagrams, Petri nets, etc.
  - ✓ Drawing by hand is a lengthy and time consuming process
  - ✓ Changes can result in having to redraw from scratch

● A data dictionary
  - ✓ A tool for storing name and representation (format) of every component of every data item

● CASE tools to combine graphical tools and data dictionaries
  - ✓ E.g., Analyst/Designer, Software through Pictures, System Architect
  - ✓ Incorporate an automatic consistency checker: Consistency between specification document and design document

● An analysis technique is unlikely to receive widespread acceptance unless a tool-rich CASE environment supports that technique

Metrics for Classical Analysis

- ✓ It is necessary to measure five fundamental metrics: Size, cost, duration, effort, and quality
- ✓ Number of pages in specification document
- ✓ Fault statistics of specification inspection
- ✓ Number of items in data dictionary

Challenges of Classical Analysis

- Resolving contradiction of specification document being simultaneously informal enough for client to understand and formal enough for development team to use as sole description of product to be built
- The boundary line between analysis ("what") and design ("how") is all too easy to cross
  - ✓ Specification document describes what to do, and not how to do it
  - ✓ List all constraints without stating how to achieve them

Comparison of Classical Analysis Techniques

Classical Analysis Method	Category	Strength	Weaknesses
Natural Language	Informal	• Easy to learn • Easy to use • Easy for the client to understand	• Imprecise • Specification can be ambiguous, contradictory or incomplete
Entity Relationship modelling	Semiformal	• Can be understood by client • More precise than informal techniques	• Not as precise as formal techniques • Cannot handle timing
Structured system Analysis			
Petrinet	Formal	• Extremely Precise • Can reduce analysis faults • Can reduce development cost and effort • Can support for correctness proving	• Hard for the development team to learn • Hard to use • Impossible for most clients to understand

2.16. DATA DICTIONARY

- Data dictionaries are generally useful when developing system models and may be used to manage all information from all types of system models.
- A data dictionary is an alphabetic list of the names included in the system models. As well as the name, the dictionary should include an associated description of the named entity and, if the name represents a composite object, a description of the composition.
- Other information such as the date of creation, the creator and the representation of the entity may also be included depending on the type of model being developed.

Name of data element	Description	Narrative
Order	Record comprising fields Order identification Customer name Customer address . . Package name Package price .	The field contain all details of an order
Order_identification	12-digit number	Unique number generated by procedure
Verify_order_ is_valid	Procedure: Input parameter:order Output parameter:no_of_error	This procedure takes order as input and check the validity of every field.

Fig. Data dictionary - keeps track of all data element

Advantages:

1. ***It is a mechanism for name management***.

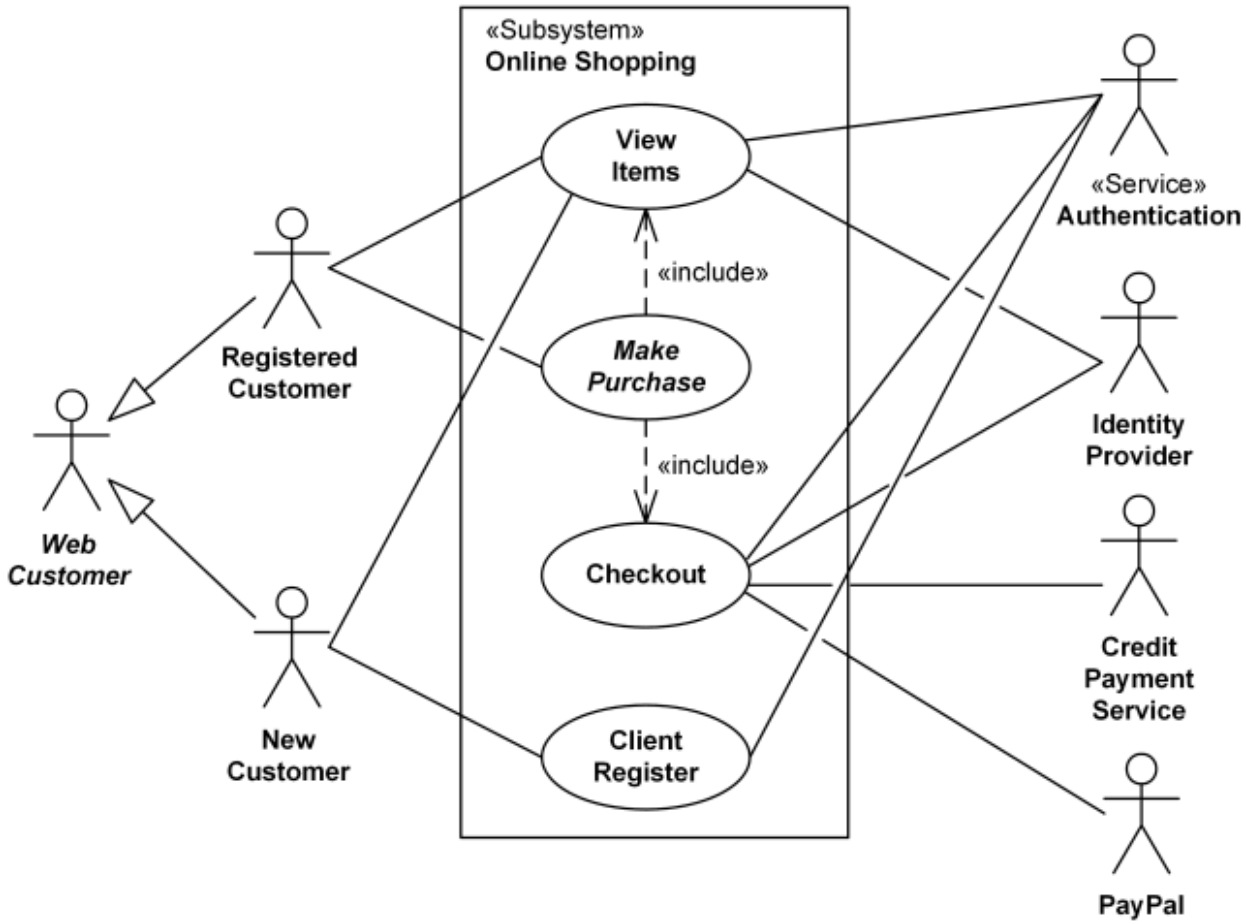
Many people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness where necessary and warn requirements analysts of name duplications.

2. ***It serves as a store of organisational information***.

As the system is developed, information that can link analysis, design, implementation and evolution is added to the data dictionary, so that all information about an entity is in one place.

PART A: 2 Marks

1. **Draw a use case diagram for an online shopping which should provide provisions for registering, authenticating the customers and also for online payment through any payment gateway like paypal. (NOV / DEC 2017)**



2. **Define Quality Function Development (QFD)? (NOV / DEC 2017)**

Quality Function Deployment (**QFD**) is a structured approach to defining customer needs or requirements and translating them into specific plans to produce products to meet those needs.

3. **Differentiate between normal and exciting requirements. (APRIL/MAY 2017)**

Normal requirements	**Exciting requirements**
Normal Requirements are what the stakeholders communicate during traditional facilitated sessions or in interviews. They cover the base functionality of the application. These requirement contribute proportionally to customer satisfaction and expectations	Exciting Requirements are aspects which the users do not expect. Often exciting requirements involve innovation of the business process or new ways of handling functionality. Stakeholder satisfaction with the application can be dramatically improved through the implementation of a few exciting requirements.

4. **What is the purpose of Data dictionaries? (APRIL/MAY 2017)**
   Data dictionaries are generally useful when developing system models and may be used to manage all information from all types of system models.

5. **What is the purpose of a Petri Net? (APRIL/MAY 2017, APRIL/MAY 2019)**
   Petri nets offer a graphical notation for stepwise processes that include choice, iteration, and concurrent execution. Petri nets have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis.
6. **What is Volatile Requirements? (APRIL/MAY 2017)**
   These are relatively stable requirements that **derive from the core activity of the organisation** and which relate directly to the domain of the system. For example, in a hospital, there will always be requirements concerned with patients, doctors, nurses and treatments.
7. **What is Elicitation? (NOV/DEC 2017)**
   Requirements elicitation is the process of *discovering, reviewing, documenting, and understanding* the user's needs and constraints for the system.
8. **List the characteristics of a good SRS. (APRIL/MAY 2016)**
   i. Correct – The SRS should be made up to date when appropriate requirements are identified.
   ii. Unambiguous – When the requirements are correctly understood then only it is possible to write an unambiguous software.
   iii. Complete – To make SRS complete,it shold be specified what a software designer wants to create software.
   iv. Consistent – It should be consistent with reference to the functionalities identified.
   v. Specific – The requirements should be mentioned specifically.
   vi. **Traceable –** What is the need for mentioned requirement? This should be correctly identified.
9. **What are the linkages between data flow and E-R Diagram? (APRIL/MAY 2016)**

   An ER diagram is the Entity Relationship Diagram, showing the relationship between different entities in a process.
   A Data Flow diagram is a symbolic structure showing how the flow of data is used in different process stages.
10. **Classify the following as functional / non-functional requirements for a banking system. (NOV/DEC 2016)**
    (a) Verifying bank balance - functional Requirements
    (b) Withdrawing money from bank - functional Requirements
    (c) Completion of transactions in less than one second – Non-functional Requirements
    (d) Extending the system by providing more tellers for customers.- functional Requirements

11. **What is a data dictionary? (NOV/DEC 2016) (NOV/DEC 2015)**

    The data dictionary can be defined as an organized collection of all the data elements of the system with precise and rigorous definitions so that user and system analyst will have a common understanding of inputs, outputs, components of stores and intermediate calculations.

12. **Define feasibility study and list the types. (NOV/DEC 2015)**
    Feasibility is defined as the practical extent to which a project can be performed successfully.
    Type of feasibility: 1. technical feasibility, 2.operational feasibility, and 3.economic feasibility

13. **What is the need for feasibility analysis? (APRIL/MAY 2015)**

A feasibility analysis evaluates the project's potential for success; therefore, perceived objectivity is an important factor in the credibility of the study for potential investors and lending institutions.

14. **How are the requirements validated? (APRIL/MAY 2015)**

While designing the user interface of software the requirement collection can be done by focusing on the profile of user who will interact with the system. Skill level, business understanding and general grasping to the new system are recorded. Users can be categorized into different categories and for each category of user requirements are elicitations.

15. What do you mean by Functional and non-functional requirement? (APRIL/MAY 2014, APR/MAY 2019)

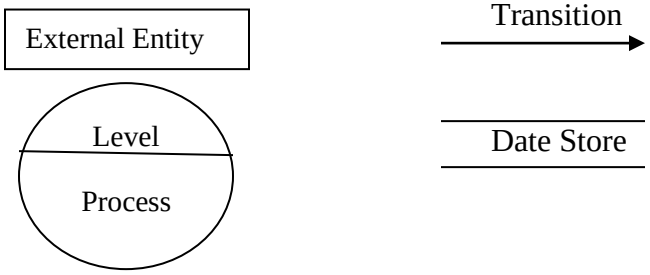
Functional requirements:

These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

Non-functional requirements

These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.

16. What is the notation used in DFD?



17. What is Petri net? NOV/DEC 2019

- Petri nets are a basic model of parallel and distributed systems. The basic idea is to describe state changes in a system with transitions.
- Petri nets — Formal technique for describing concurrent interrelated activities

18. What are non-functional requirements? NOV/DEC 2019

Nonfunctional requirements are the characteristics of the system which cannot be expressed as functions - such as the maintainability, portability, usability, robustness, ease of use of the system.

PART B – ANNA UNIVERSITY QUESTIONS

1. What is feasibility study? How it helps in requirement engineering process? **NOV / DEC 2017**
2. How will you classify the requirement types for a project, give examples**. NOV / DEC 2017**
3. List the stake holders and all types of requirements for an online train reservation system **NOV / DEC 2017**
4. Consider the process of ordering a pizza over the phone. Draw the usecase diagram and also sketch the activity diagram representing each step of the process, from the moment you pick the phone to the point where you start eating the pizza. Include activities that others need to perform. Add exception handling to the activity diagram you developed. Consider at least two exceptions (e.g. delivery person wrote down wrong address, deliver person brings wrong pizza) **NOV / DEC 2017**
5. What is requirement engineering? Explain in detail the various processes in requirements engineering. **APRIL / MAY 2017, NOV/DEC 2019**
6. Explain the feasibility studies. What are the outcomes? Does it have implicit or explicit effects on software requirement collection? **APRIL / MAY 2017**
7. Write a note on what are the difficulties in elicitation, requirement elicitation. **APRIL / MAY 2017, APRIL/MAY 2017**
8. Explain the organization of SRS and highlight the importance of each subsection. **MAY /JUNE 2016, APRIL/MAY 2017**
9. Requirements analysis is unquestionably the most communication intensive step in the software engineering process. Why does the communication path frequently breaks down? **MAY /JUNE 2016**
10. Differentiate between user and system requirements. **MAY /JUNE 2016**
11. Describe the requirements change management process in detail. **MAY /JUNE 2016**
12. What is requirements elicitation? Briefly describe the various activities performed in requirements elicitation phase with an example of a watch system that facilitates to set time and **alarm. NOV / DEC 2016**
13. Explain the software requirement engineering process with neat diagram. **NOV / DEC 2015**
14. Draw Use Case and Data Flow diagrams for a Restaurant System. The activities of the Restaurant system are listed below.
    Receive the Customer food Orders. Produce the customer ordered foods, Serve the customer with their ordered foods, Collect Payment from customers, Store customer payment details, Order Raw Materials for food products, Pay for Raw Materials and Pay for Labor. **NOV / DEC 2015**
15. Draw a Petri Net that depicts the operation of an "Automated Teller Machine" State the functional requirements you are considering. **NOV/DEC 2019**

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. INTRODUCTION

- Software design encompasses *the set of principles, concepts, and practices that lead to the development of a high-quality system or product*.
- Design creates representation or model of the software. Design model provides detail about *software architecture, data structure, interfaces and components* that are necessary to implement the system.
- Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used.
- Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

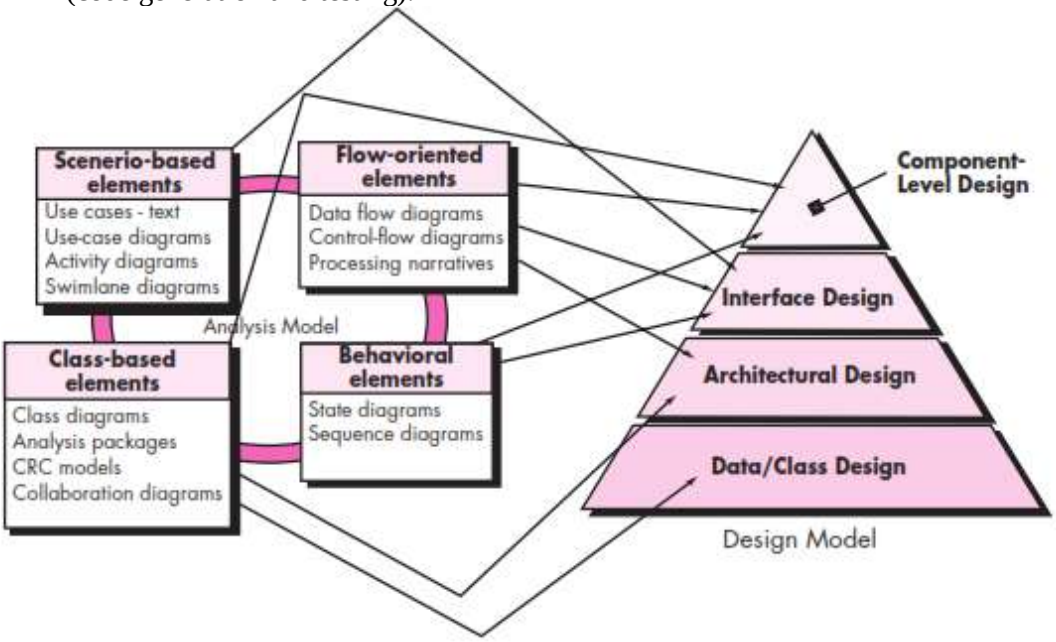


Figure. Translating the requirements model into the design model

Four design models required for a complete specification of design

1) **Data/class design**
   The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.
   The objects and relationships defined in the CRC(class responsibility collaborator) diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action.
2) **Architectural design**
   The architectural design defines the **relationship between major structural elements** of the software, **the architectural styles and design patterns** and the **constraints** that affect the way in which architecture can be implemented.
3) **Interface design**
   The interface design describes how the software communicates with systems that **interoperate with it, and with humans who use it**. An interface implies a flow of information and a specific type of behavior. Therefore, **usage scenarios and behavioral models** provide much of the information required for interface design.
4) **Component-level design**
   The component-level design *transforms structural elements of the software architecture into a procedural description* of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

3.2. DESIGN PROCESS:

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

Software Quality Guidelines and Attributes

Three characteristics that serve as a guide for the evaluation of a good design: **(or) goals of good design**

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

Quality Guidelines

1. A design should exhibit an architecture that
   a. Has been *created using recognizable architectural styles or patterns*,
   b. Is composed of *components that exhibit good design characteristics*
   c. Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2. A design should be **modular**; that is, the software should be **logically partitioned** into elements or subsystems.

3. A design should contain *distinct representations of data, architecture, interfaces, and components*.

4. A design **should lead to data structures that are appropriate for the classes** to be implemented and are drawn from recognizable data patterns.

5. A design **should lead to components** that exhibit independent functional characteristics.

6. A design **should lead to interfaces that reduce the complexity of connections** between components and with the external environment.

7. A design should be **derived using a repeatable method** that is driven by information obtained during software requirements analysis.

8. A design **should be represented using a notation** that effectively communicates its meaning.

Quality Attributes.

A set of software quality attributes that has been given the acronym **FURPS**—functionality, usability, reliability, performance, and supportability.

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

- **Functionality**is assessed by evaluating the *feature set and capabilities* of the program, the generality of the functions that are delivered, and the security of the overall system.
- **Usability**is assessed by considering human factors, overall *aesthetics, consistency, and documentation*.
- **Reliability** is evaluated by measuring the *frequency and severity of failure*, the *accuracy of output results*, the *mean-time-to-failure* (MTTF), the ability to *recover* from failure, and the *predictability* of the program.
- **Performance** is measured by considering *processing speed, response time, resource consumption, throughput, and efficiency*.
- **Supportability** combines the ability to extend the program (*extensibility*), *adaptability, serviceability*—these three attributes represent a more common term, maintainability—and in addition, *testability, compatibility, configurability* (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized.
- Not every software quality attribute is weighted equally as the software design is developed.
- One application may stress functionality with a special emphasis on security.
- Another may demand performance with particular emphasis on processing speed.
- A third might focus on reliability.
- Regardless of the weighting, it is important to note that these *quality attributes must be considered as design commences*, not after the design is complete and construction has begun.

3.3. DESIGN CONCEPTS:

Design creates a representation or model of the software, the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system. Fundamental software design concepts **provide the necessary framework for "getting it right".**

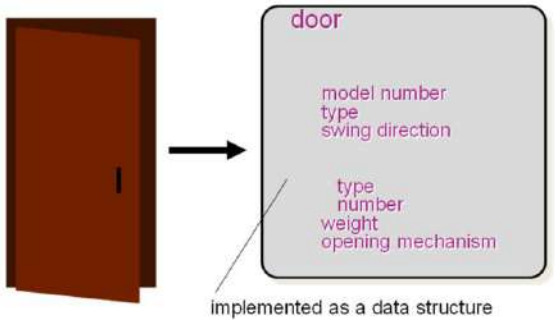
Important software design concepts

1. Abstraction
2. Architecture
3. Patterns
4. Separation of Concerns
5. Modularity
6. Information Hiding
7. Functional Independence
8. Refinement
9. Aspects
10. Refactoring
11. Object-Oriented Design Concepts
12. Design Classes

1) Abstraction

"Abstraction permits one to concentrate on a problem at some level of abstraction without regard to low level details"

- Procedural Abstraction
  - Sequence of instructions that have a specific and limited function.
  - Instructions are given in a named sequence
  - Each instruction has a limited function
  - The name of a procedural abstraction implies these functions, but specific details are suppressed.
  - An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)
- Data Abstraction
  - This is a named collection of data that describes a data object.
  - Data abstraction includes a set of attributes that describe an object.
  - The data abstraction for door would encompass set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.



- Control Abstraction
  - A program control mechanism without specifying internal details, e.g., semaphore, rendezvous

2) Architecture

Architecture is the *structure or organization of program components* (modules), the manner in which these components interact, and the structure of data that are used by the components. Components can be generalized to represent major system elements and their interactions.

Desired properties of an architectural design

- Structural Properties

- This defines the components of a system and the manner in which these interact with one another.

- Extra Functional Properties
  - This addresses how the design architecture achieves requirements for performance, reliability, capacity, adaptability, and security
- Families of Related Systems
  - The ability to reuse architectural building blocks

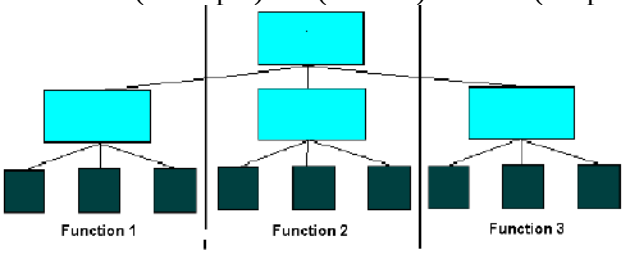
Kinds of Models

1) **Structural models**: represent architecture as an organized collection of components.
2) **Framework models**: increase the level of design abstraction by identifying repeatable architecture design frameworks (patterns)
3) **Dynamic models**: address the behavior aspects of the program architecture
4) **Process models**: focus on the design of the business or technical process
5) **Functional models**: can be used to represent the functional hierarchy of a system

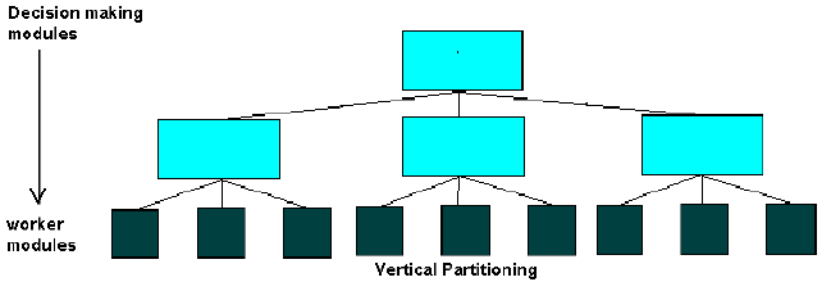
Program Structure Partitioning

- **Horizontal Partitioning**
  - Easier to test
  - Easier to maintain (questionable)
  - Propagation of fewer side effects (questionable)
  - Easier to add new features

F1 (Ex: Input) F2 (Process) F3(Output)



- **Vertical Partitioning**
  - Control and work modules are distributed top down
  - Top level modules perform control functions
  - Lower modules perform computations
    - Less susceptible to side effects
    - Also very maintainable



3) Pattern

- A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine
(1) Whether the pattern is applicable to the current work,
(2) Whether the pattern can be reused (hence, saving design time), and
(3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4) Separation of Concerns

- ❖ Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- ❖ A concern is a feature or behavior that is specified as part of the requirements model for the software.

- ❖ By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- ❖ For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.
- ❖ It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy

5) Modularity

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

- Follows "divide and conquer" concept, a complex problem is broken down into several manageable pieces
- Let p_1 and p_2 be two program parts, and E the effort to solve the problem. Then,
  $E(p_1+p_2) > E(p_1)+E(p_2)$, often >>
- A need to divide software into optimal sized modules.
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding more difficult.

Modularity & Software Cost

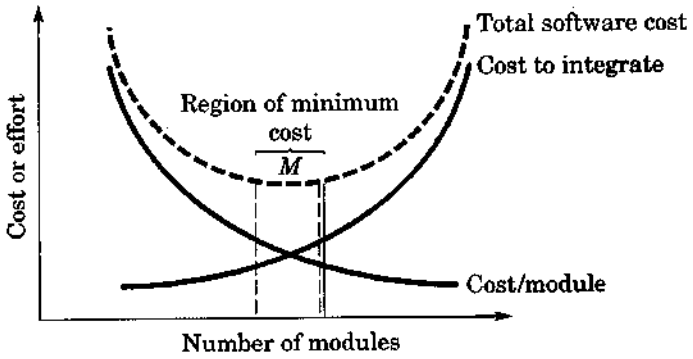


FIGURE 13.2. Modularity and software cost

Objectives of modularity in a design method

- Modular Decomposability
  - Provide a systematic mechanism to decompose a problem into sub problems
- Modular Composability
  - Enable reuse of existing components to be assembled into a new system
- Modular Understandability
  - Can the module be understood as a stand alone unit? Then it is easier to understand and change.
- Modular Continuity
  - If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of the side effects is reduced
- Modular Protection
  - If there is an error in the module, then those errors are localized and not spread to other modules

Benefits of modularize a design

- Development can be more easily planned;
- Software increments can be defined and delivered;
- Changes can be more easily accommodated;
- Testing and debugging can be conducted more efficiently,
- Long-term maintenance can be conducted without serious side effects.

6) Information Hiding

- Modules are characterized by design decisions that are hidden from others. Modules should be specified and designed so that information (algorithms and data) contained within a module **is inaccessible to other modules** that have no need for such information.
- Modules communicate only through well defined interfaces

- Enforce access constraints to local entities and those visible through interfaces
- Very important for accommodating change and reducing coupling.
- Abstraction helps to define the procedural (or informational) entities that make up the software.
- Hiding **defines and enforces access constraints** to both procedural detail within a module and any local data structure used by the module

Benefits Information Hiding:

- Inadvertent errors introduced during modification are less likely to propagate
- Reduces the likelihood of "side effects"
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

7) Functional Independence

- Functional independence is achieved by developing modules with "**singleminded" function and an "aversion" to excessive interaction** with other modules.
- Each module **addresses a specific subset of requirements** and has a simple interface when viewed from other parts of the program structure.
- Critical in dividing system into independently implementable parts
- Measured by two qualitative criteria
  - **Cohesion** : Relative **functional strength** of a module
  - **Coupling** : **Relative interdependence** among modules

Modular Design – Cohesion

- A cohesive module performs a single task requiring little interaction with other components in other parts of a program.
- Different levels of cohesion
  - Coincidental, logical, temporal, procedural, communications, sequential, functional

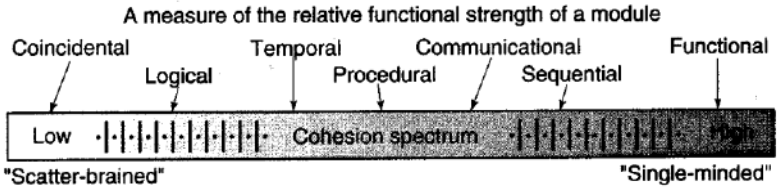


FIGURE 13.6.
Cohesion

- Coincidental Cohesion
  - The parts of a component are not related but simply bundled into a single component.
  - Harder to understand and not reusable
- Logical Cohesion
  - Similar functions such as input, error handling, etc. put together. Functions fall in same logical class. May pass a flag to determine which ones executed.
  - Interface difficult to understand. Code for more than one function may be intertwined, leading to severe maintenance problems.
  - Difficult to reuse
- Temporal Cohesion
  - All of statements activated at a single time, such as start up or shut down, are brought together. Initialization, clean up.
  - Functions weakly related to one another, but more strongly related to functions in other modules so may need to change lots of modules when do maintenance.
- Procedural cohesion:
  - A single control sequence, e.g., a loop or sequence of decision statements. Often cuts across functional lines. May contain only part of a complete function or parts of several functions.
  - Functions still weakly connected, and again unlikely to be reusable in another product.

- Communicational cohesion:
  - Operate on same input data or produce same output data. May be performing more than one function. Generally acceptable if alternate structures with higher Cohesion cannot be easily identified.
  - Still problems with reusability.
- Sequential cohesion:
  - Output from one part serves as input for another part. May contain several functions or parts of different functions.
- Informational cohesion:
  - Performs a number of functions, each with its own entry point, with independent code for each function, all performed on same data structure. Different than logical cohesion because functions not intertwined.
- Functional cohesion:
  - Each part necessary for execution of a single function. e.g., compute square root or sort the array.
  - Usually reusable in other contexts. Maintenance easier.
- Type cohesion:
  - Modules that support a data abstraction.
  - Not strictly a linear scale. Functional much stronger than rest while first two much weaker than others. Often many levels may be applicable when considering two elements of a module. Cohesion of module considered as highest level of cohesion that is applicable to all elements in the module.

Modular Design – Coupling

- Coupling describes the **interconnection among modules**
- Coupling depends on the **interface complexity between modules**, the point at which entry orreference is made to a module, and what data pass across the interface.

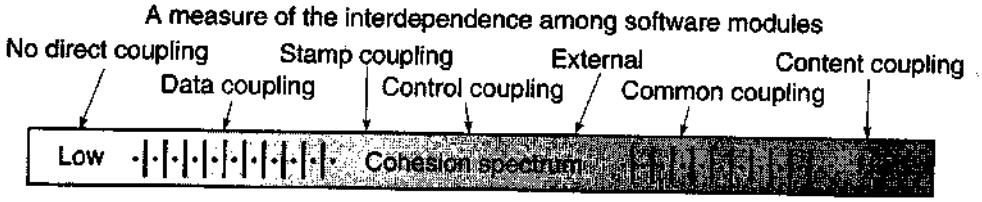


FIGURE 13.7.
Coupling

- Data coupling
  - Occurs when **one module passes local data values to another** as parameters
- Stamp coupling
  - Occurs when **part of a data structure is passed** to another module as a parameter
  - similar to common coupling except that global variables are shared selectively among routines that require the data. E.g., packages in Ada. More desirable than common coupling because fewer modules will have to be modified if a shared data structure is modified. Pass entire data structure but need only parts of it.
- Control Coupling
  - Occurs when **control parameters are passed** between modules. So that one module controls the sequence of processing steps in another module
- Common Coupling
  - Occurs when multiple modules **access common data areas** such as Fortran Common or C extern
- Content Coupling
  - If one module directly references the contents of the other.
  - When one module modifies local data values or instructions in another module.
  - If one refers to local data in another module.
  - If one branches into a local label of another.
- Subclass Coupling
  - The coupling between a class and its parent class

Examples of Coupling

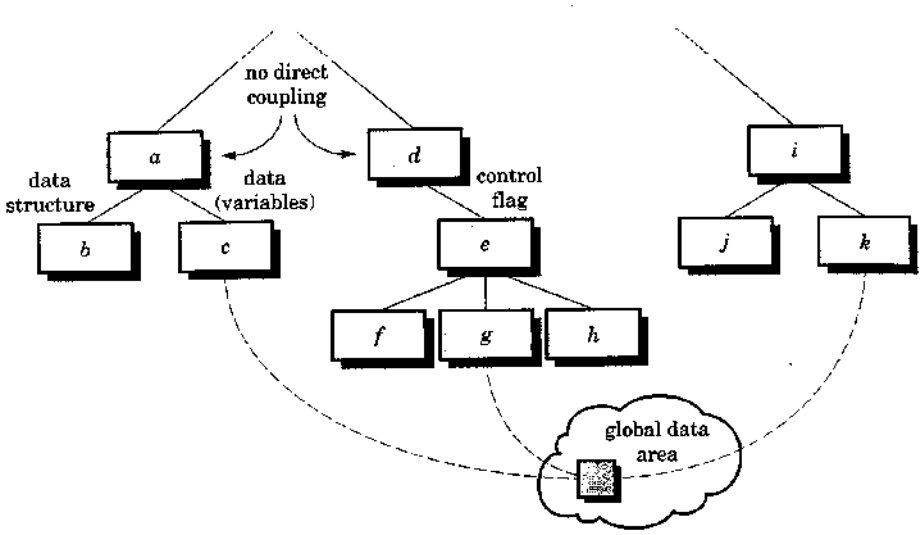


FIGURE 13.8. Types of coupling

Different between Cohesion and Coupling

Cohesion	Coupling
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** strength.	Coupling shows the relative **independence** among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the **single** thing.	Coupling is a degree to which a component / module is connected to the **other** modules.
While designing you should strive for **high cohesion** i.e. a cohesive component/ module focus on a single task (i.e., **single-mindedness**) with little interaction with other modules of the system.	While designing you should strive for **low coupling** i.e. **dependency** between modules should be less.
Cohesion is the kind of natural extension of data hiding for example, **class** having all members visible with a package having default visibility.	Making private fields, private methods and non public classes provides loose coupling.
Cohesion is **Intra – Module** Concept.	Coupling is **Inter -Module** Concept.

8) Refinement

- Refinement is actually a process of elaboration.
- Refinement is a process where one or several instructions of the program are decomposed into more detailed instructions.
- Begin with a statement of function (or description of information) that is defined at a high level of abstraction and then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Refinement helps to reveal low-level details as design progresses.
- Stepwise refinement is a top down strategy
  - Basic architecture is developed iteratively
  - Step wise hierarchy is developed

9) Aspects

- An aspect is a representation of a crosscutting concern.
- For example, generic security requirement that states that a registered user must be validated prior to using an application. This requirement is applicable for all functions that are available to registered users of the system.
- The design representation, of the requirement a registered user must be validated prior to using the system, is an aspect of the system.

- An aspect is implemented as a **separate module (component)** rather than as software fragments that are "scattered" or "tangled" throughout many components
- The design architecture should support a **mechanism for defining an aspect**—a module that enables the **concern to be implemented across all other concerns** that it crosscuts.

10) Refactoring

- "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
- Refactoring is a **reorganization technique** that simplifies the design (or code) of a component **without changing its function or behavior**.

- When software is refactored, the existing design is examined for
  - Redundancy
  - Unused design elements
  - Inefficient or unnecessary algorithms
  - Poorly constructed or inappropriate data structures or any other design failure that can be corrected to yield a better design.

11) Object-Oriented Design Concepts

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

12) Design class

- The requirements model defines a set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.
- As the design model evolves, we will define a set of design classes that refine the analysis classes by providing **design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution**.

Five different types of design classes

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

Four characteristics of a well-formed design class:

1) Complete and sufficient.

- A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class.

2) Primitiveness.

- **Methods** associated with a design class should be focused on accomplishing **one service** for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

3) High cohesion

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

4) Low coupling.

- Within the design model, it is necessary for design classes to collaborate with one another. However, **collaboration should be kept to an acceptable minimum**.
- This restriction, called the **Law of Demeter**, suggests that a method should only send messages to methods in neighboring classes.

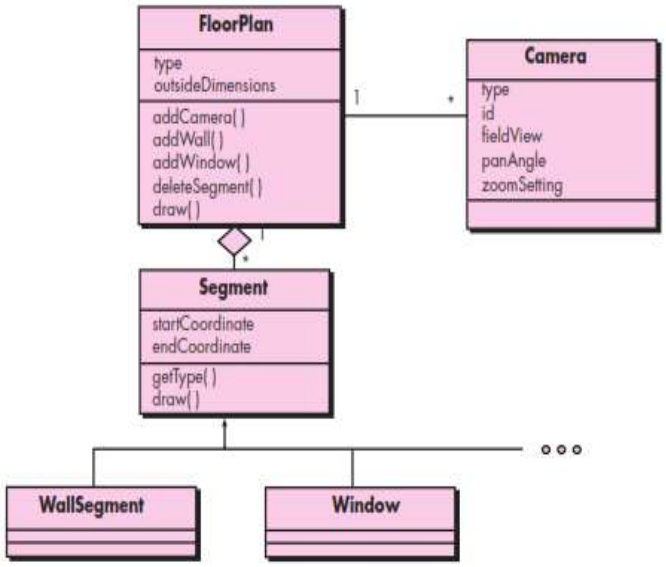


Figure. Design class for FloorPlan and composite aggregation for the class

3.4 . DESIGN MODEL

The design model can be viewed in two different dimensions:

(1) The **process dimension**indicates the **evolution of the design model** as design tasks are executed as part of the software process.

(2) The **abstraction dimension**represents **the level of detail** as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

- The elements of the design model use many of the same UML diagrams that were used in the analysis model.
- The difference is that these **diagrams are refined and elaborated** as part of design;
- More implementation-specific detail is provided, and
- Architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

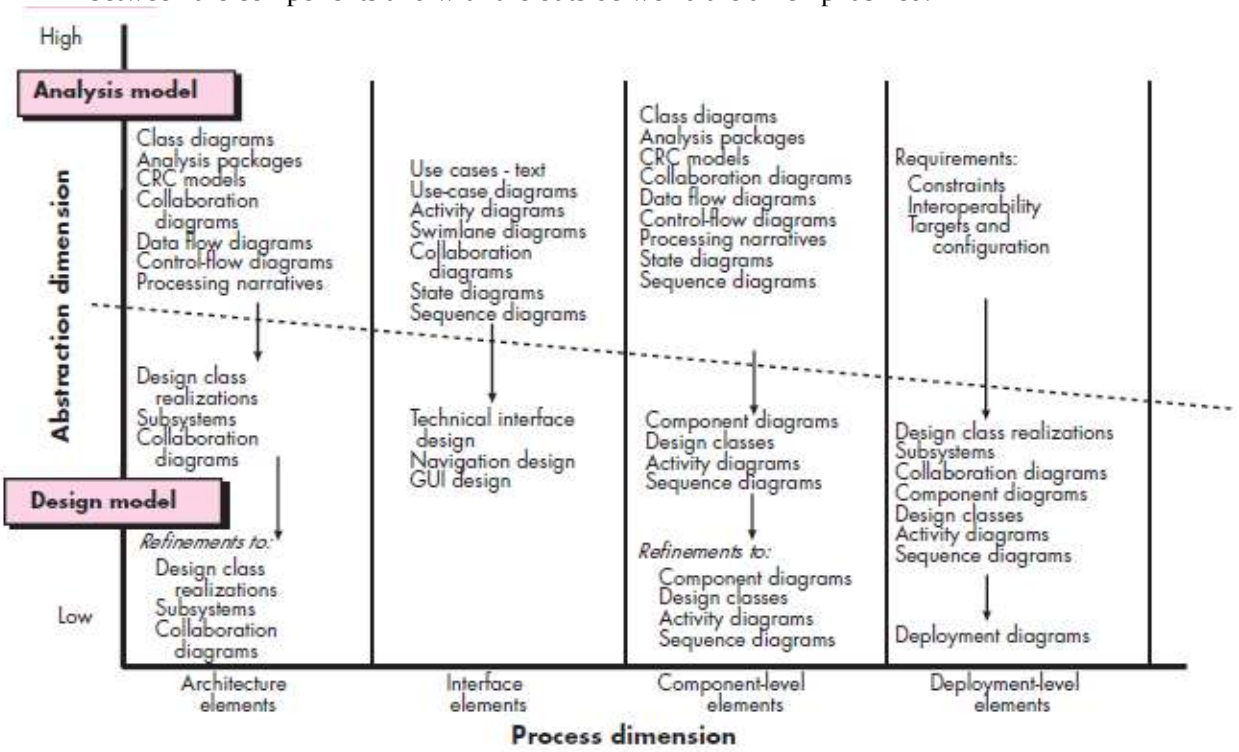


Fig. Dimension of design model

1. Data Design Elements:

- Data design creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

At the program component level:- design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
At the application level:- translation of a data model into a database is pivotal to achieving the business objectives of a system.
At the business level:- the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

2. Architectural Design Elements:
The architectural 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;
(3) The availability of architectural styles and patterns

- The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture.

3. Interface Design Elements:
The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
An interface is a set of operations that describes some part of the behavior of a class and provides access to these operations.
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;
(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.
- **Usability design incorporates aesthetic elements** (e.g., layout, color, graphics, interaction mechanisms), **ergonomic elements** (e.g., information layout and placement, metaphors, UI navigation), and **technical elements** (e.g., UI patterns, reusable components
- The design of **external interfaces** requires definitive information about the **entity to which information is sent or received**.
- The design of external interfaces should incorporate **error checking and (when necessary) appropriate security features.**
- The design of **internal interfaces** is closely aligned with component-level design. Design realizations of analysis classes represent all operations and the messaging schemes required to **enable communication and collaboration between operations in various classes**.

EXAMPLE:

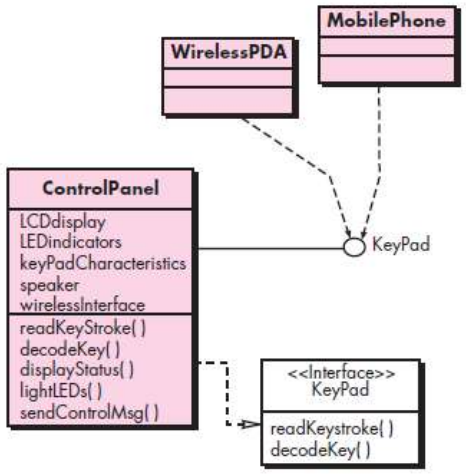


Fig. Interface representation for Control- Panel

4. Component-Level Design Elements:

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

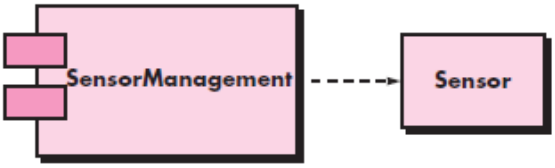


Fig. A UML component diagram

- The **SensorManagement**component performs all functions associated with *SafeHome*sensors including monitoring and configuring them.

5. Deployment-Level Design Elements:

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

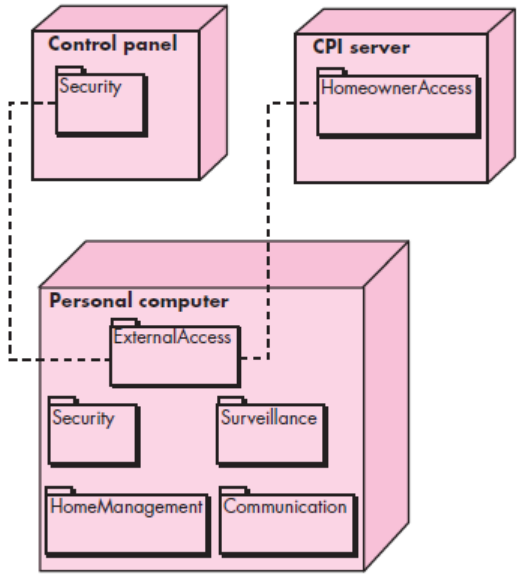


Fig. A UML deployment diagram

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. (providing Internet-based access to the system).

- In the figure, three computing environments are shown (in actuality, there would be more including sensors, cameras, and others).
- The subsystems (functionality) housed within each computing element are indicated. For example, the personal computer houses subsystems that implement security, surveillance, home management, and communications features.
- In addition, an external access subsystem has been designed to manage all attempts to access the *SafeHome*system from an external source. Each subsystem would be elaborated to indicate the components that it implements.
- The diagram shown in Figure is in ***descriptor form***. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details. For example, the "personal computer" is not further identified. It could be a Mac or a Windows-based PC, a Sun workstation, or a Linux-box. These details are provided in ***instance form***.

3.5. DESIGN HEURISTIC

The program structure can be manipulated according to the following set of heuristics:

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

- Once the program structure has been developed, modules may be **exploded or imploded** with an eye toward improving module independence. An exploded module becomes two or more modules in the final program structure.
- An imploded module is the result of combining the processing implied by two or more modules. An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

2. Attempt to **minimize structures with high fan-out; strive for fan-in as depth increases**. The structure shown inside the cloud in figure does not make effective use of factoring. All modules are "pancaked" below a single control module. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

3.Keep the scope of effect of a module within the scope of control of that module. The scope of effect of module e is defined as all other **modules that are affected by a decision made in module e**. The scope of control of module e is **all modules that are subordinate** and ultimately subordinate to module e

4. Evaluate module interfaces to reduce complexity and redundancy and improve consistency. Module interface complexity is a prime cause of software errors .Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly unrelated data passed via an argument list or other technique) is an indicationof low cohesion. The module in question should be reevaluated.

5. Define modules whose function is predictable, but avoid modules that are overly restrictive.

- A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use.
- A module that restricts processing to a single subfunction exhibits high cohesion and is viewed with favor by a designer. However, a module that arbitrarily restricts the size of a local data structure, options within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

6. Strive for "controlled entry" modules by avoiding "pathological connections." This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. Pathological connection refers to branches or references into the middle of a module.

Effective Modular Design

1. **Information hiding**:
   Modules should be specified and designed so that the internal details of modules should be invisible or inaccessible to other modules.
   Major benefits:reduce the change impacts in testing and maintenance
2. **Functional independence:**
   Design modules based on independent functional features
   Major benefits*:*effective modularity
3. **Cohesion:a natural extension of the information hiding concept**
   - A cohesive module performs a single task requiring little interaction with other components in other parts of a program.
   - Different levels of cohesion
     - Coincidental, logical, temporal, procedural, communications, sequential, functional
   - Coincidental Cohesion
     - Occurs when modules are **grouped together for no reason** at all
   - Logical Cohesion
     - Modules have a logical cohesion, but no actual connection in data and control
   - Temporal Cohesion

- Modules are bound together because they must be used at **approximately the same time**
- Communication Cohesion
  - Modules grouped together because they access the **same Input/Output devices**
- Sequential Cohesion
  - Elements in a module are linked together by the necessity to be activated in a **particular order**
- Functional Cohesion
  - All elements of a module relate to the performance of a single function

4. Coupling

- Coupling describes the **interconnection among modules**
- Coupling depends on the **interface complexity between modules**, the point at which entry orreference is made to a module, and what data pass across the interface.

Types of coupling

- Data coupling
  - Occurs when **one module passes local data values to another** as parameters
- Stamp coupling
  - Occurs when **part of a data structure is passed** to another module as a parameter
- Control Coupling
  - Occurs when **control parameters are passed** between modules
- Common Coupling
  - Occurs when multiple modules **access common data areas** such as Fortran Common or C extern
- Content Coupling
  - if one module directly references the contents of the other
- Subclass Coupling
  - The coupling between a class and its parent class

3.6. ARCHITECTURAL DESIGN-INTRODUCTION

- Architectural design represents the structure of data and program components that are required to build a computer-based system.

It considers

- The **architectural style** that the system will take
- The **structure and properties** of the components that constitute the system
- The **interrelationships** that occur among all architecturalcomponents of a system.

Software Architecture:

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

The architecture is not the operational software. Rather, it is a representation that enables you to

(1) **Analyze the effectiveness** of the design in meeting its stated requirements,

(2) **Consider architectural alternatives** at a stage when making design changes is still relatively easy

(3) **Reduce the risks** associated with the constructionof the software.

Difference between the terms architecture and design:

- A design is aninstance of an architecture similar to an object being an instance of a class.
- For example,consider the client-server architecture. I can design a network-centric software system inmany different ways from this architecture using either the Java platform (Java EE) orMicrosoft platform (.NET framework). So, there is one architecture, but many designs canbe created based on that architecture. Therefore, you cannot mix “architecture” and“design” with each other.

Design of software architecture considers two levels of the design pyramid

1) Data design
2) Architectural design.

Data design enables you to represent the data component of the architecture in conventional systems and class definitions (encompassing attributes and operations) in object-oriented systems.

Architectural design focuses on therepresentation of the structure of software components, their properties, and interactions.

3.7. ARCHITECTURAL STYLES

- The software that is built for computer-based systems also exhibits one of manyarchitectural styles. Each style describes a system category that encompasses

(1) A set of components (e.g., a database, computational modules) that perform a functionrequired 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;

(4) Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its parts.

- 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.

A pattern differs from a style in a number of fundamental ways:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;

(2) A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency);

(3) Architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts). Patterns can be used in conjunction with an architectural style to shape theoverall structure of a system.

ARCHITECTURAL STYLES ARE:

1) **Data-centered architectures.**

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

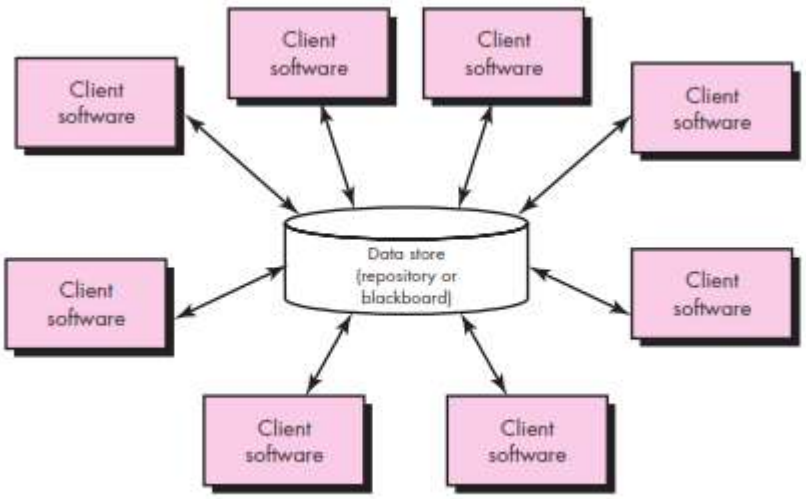


Figure.Data-centered architecture

- Data-centered architectures promote integrability. That is, existingcomponents can be changed and new client components added to the architecturewithout concern about other clients.
- Data can be passed among clients using the blackboard mechanism. Client components independently executeprocesses.

2) **Data-flow architectures.**

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the nextfilter) of a specified form.

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

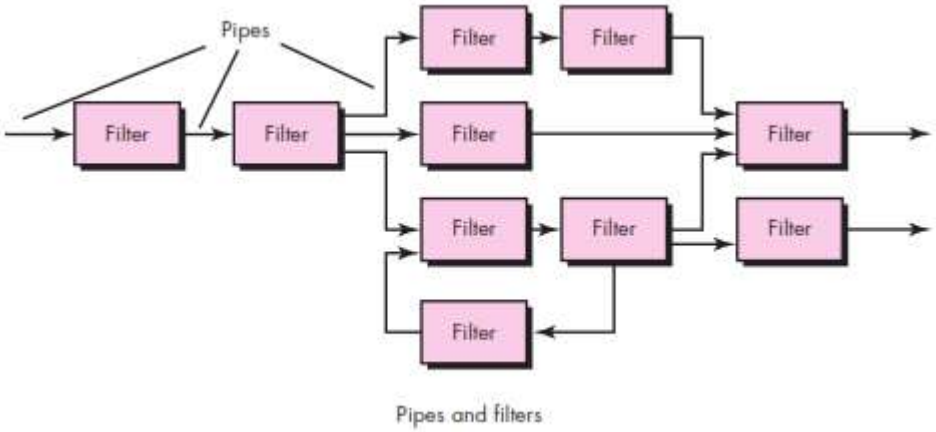


Figure. Data-flow architecture

3) Call and return architectures.

- This architectural style enables you to achieve aprogram structure that is relatively easy to modify and scale. A number of substyles exist within this category:
- Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Figure illustrates architecture of this type.
- Remote procedure call architectures. The components of mainprogram/subprogram architecture are distributed across multiple computerson a network.

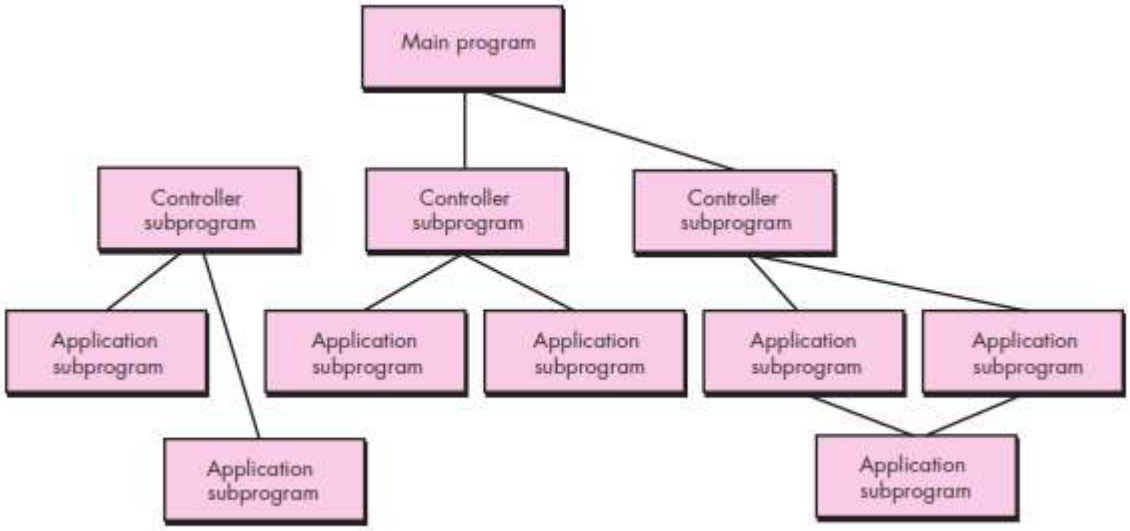


Figure. Main program/subprogram architecture

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.

5) Layered architectures.

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

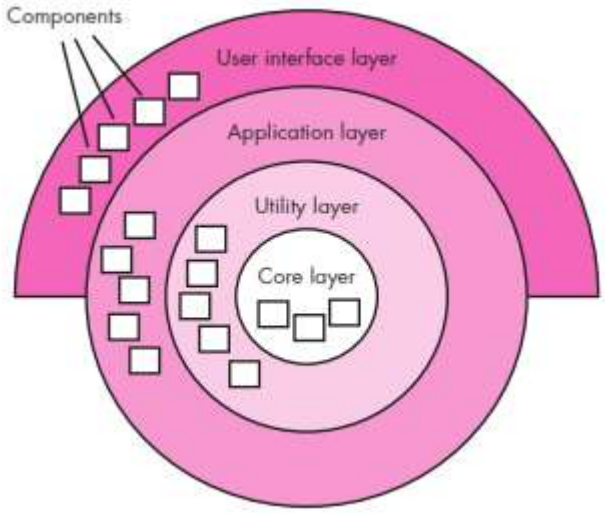


Figure. Layered architecture

- These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen.

Architectural Patterns

- 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.Most applications fit within a specific domain and that one or more architectural styles may be appropriate for that genre.
- For example, the overall architectural style for an application might be call-and return or object-oriented. But within that style, you will encounter a set of commonproblems that might best be addressed with specific architectural patterns.

3.8 . ARCHITECTURAL DESIGN

- As architectural design begins, the software to be developed must be put into **context**—that is, the design should define the external entities (other systems, devices,people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.
- An archetype is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed,but the archetypes themselves do not provide enough implementation detail.
- Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

3.8.1. 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. Systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

• **Superordinate systems**—those systems that use the target system as part of some higher-level processing scheme.

• **Subordinate systems**—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

• **Peer-level systems**—those systems that interact on a peer-to-peer basis (i.e.,information is either produced or consumed by the peers and the targetsystem.

• **Actors—**entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

- Each of these external entities communicates with the target system through an interface(the small shaded rectangles).
- To illustrate the use of the ACD, consider the home security function of the SafeHomeproduct. The overall SafeHomeproduct controller and the Internet-basedsystem are both superordinate to the security function .The surveillance function is a peer system and uses (is used by)the home security function in later versions of the product.
- The homeowner and controlpanels are actors that are both producers and consumers of informationused/produced by the security software. Finally, sensors are used by the securitysoftware and are shown as subordinate to it.

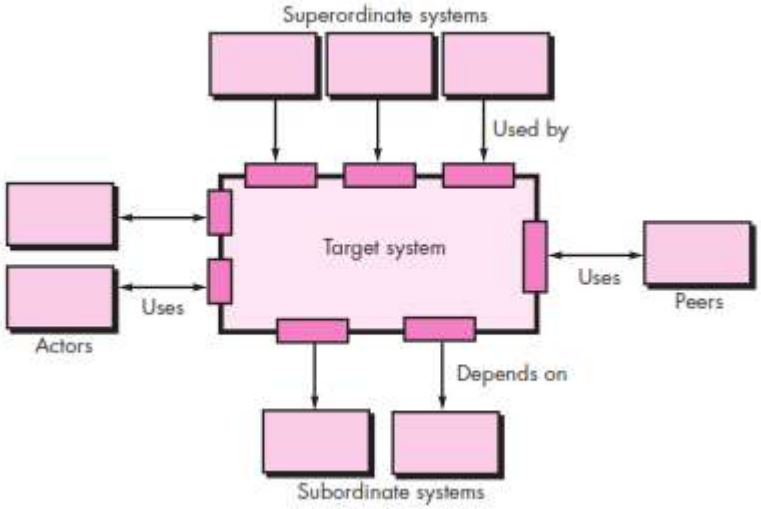


Figure. Architectural context diagram

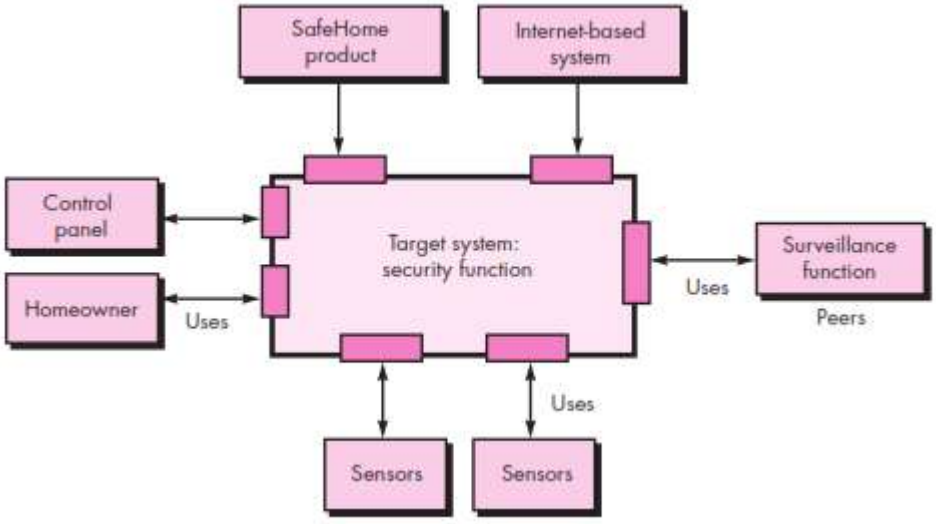


Figure. Architectural context diagram for the SafeHomesecurity function

- As part of the architectural design, the details of each interface would have to be specified. All data that flow into and out of the target systemmust be identified at this stage.

3.8.2. Defining Archetypes:

- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems.
- The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.
- In many cases, archetypes can be derived by examining the analysis classes definedas part of the requirements model. SafeHomehome security function archetypes are:

• **Node.** Represents a cohesive collection of input and output elements ofthe 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 feedsinformation into the target system.

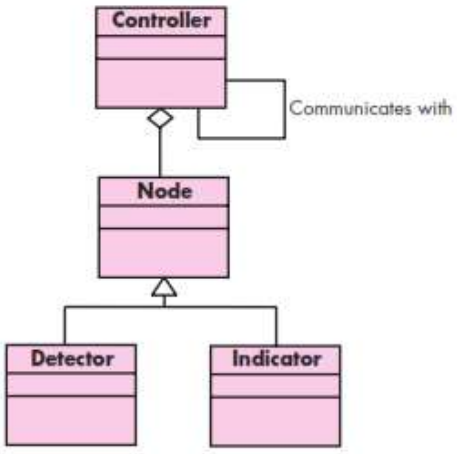


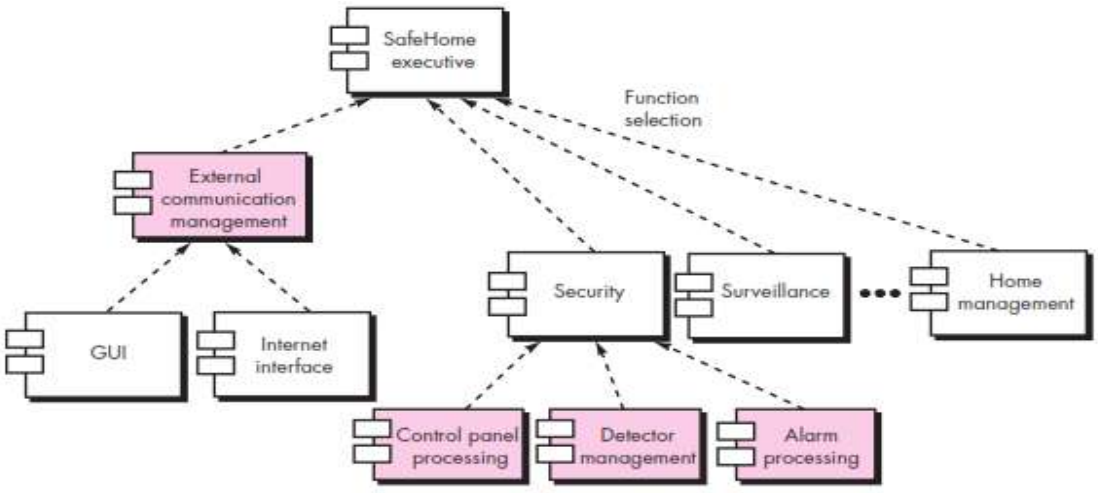
Figure.UML relationships for SafeHomesecurity function archetypes

• **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren,flashing lights, bell) for indicating that an alarm condition is occurring.

• **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they havethe ability to communicate with one another.

3.8.3. Refining the Architecture into Components:

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



❖ The control panel processing component interacts with the homeowner to arm/disarm the security function. The detector management component polls sensors to detect an alarm condition, and the alarm processing component produces output when an alarm is detected.

3.8.4. Describing Instantiations of the System:

- The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components havebeen identified. However, further refinement isstill necessary.
- To accomplish this, an actual instantiation of the architecture is developed. By this, the architecture is applied to a specific problem with the intent of demonstratingthat the structure and components are appropriate.
- For example, the detector management component interacts with a scheduler infrastructure component that implements polling of each sensor object used by the security system. Similar elaboration is performed for each of the components represented in Figure.

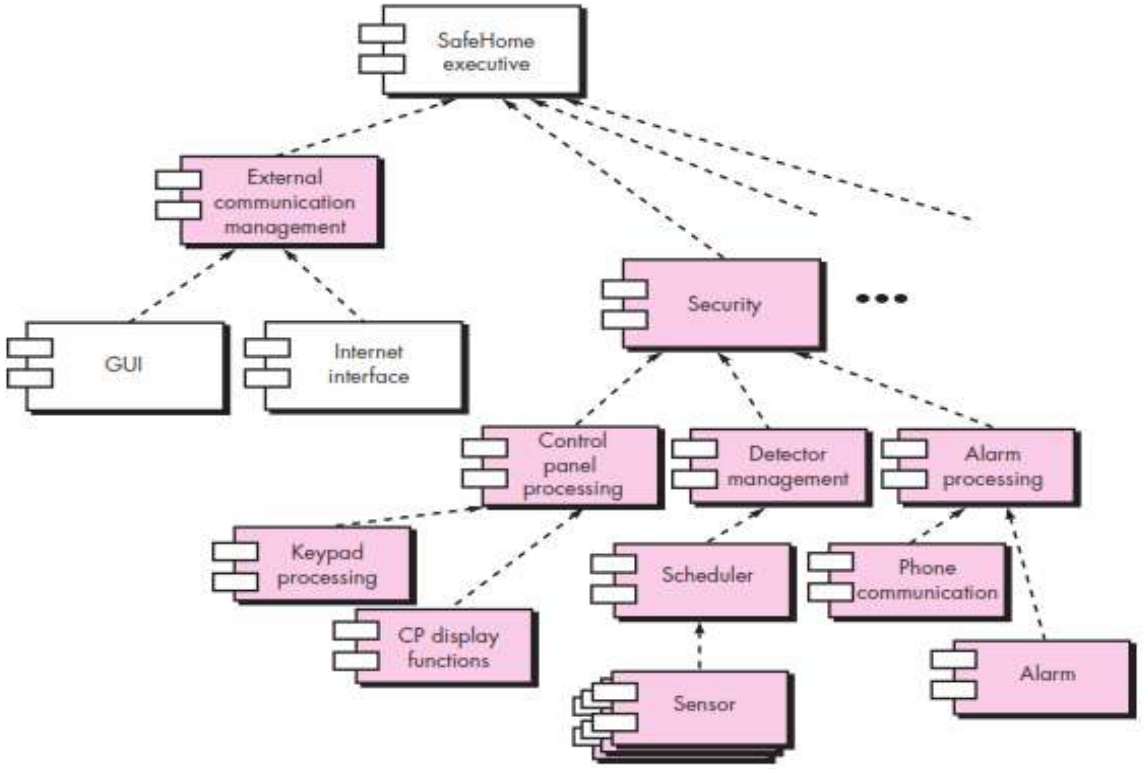


Figure. An instantiation of the security function with component elaboration

3.9. ARCHITECTURAL MAPPING USING DATA FLOW

- The architectural styles represent radically different architectures.So it should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist.
- In fact, there is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in using the techniques.
- To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems.
- The call and return architecture can reside within other more sophisticated architectures. For example, the architecture of one or more components of client-server architecture might be call and return.
- **A mapping technique, called structured design, is often characterized as adata flow-oriented design method because it provides a convenient transition from adata flow diagram to software architecture.**
- **The transition from information flow (represented as a DFD) to program structure is accomplished as part of a sixstepprocess:**

(1) The type of information flow is established
(2) Flow boundaries are indicated
(3) The DFD is mapped into the program structure
(4) Control hierarchy is defined
(5) The resultant structure is refined using design measures and heuristics, and
(6) The architectural description is refined and elaborated.

- **In transaction flow**, a single data item, called a *transaction*, causes the data flow to branch along one of a number of flow paths defined by the nature of the transaction.

TRANSFORM MAPPING:

Transform mapping is a set of design steps that allows a DFD with transform flowcharacteristics to be mapped into a specific architectural style. To illustrate this approach, we again consider the SafeHomesecurity function. One element of theanalysis model is a set of data flow diagrams that describe information flow within the security function. To map these data flow diagrams into a software architecture,you would initiate the following design steps:

Step 1. Review the fundamental system model.

- The fundamental system modelor context diagram depicts the security function as a single transformation, representingthe external producers and consumers of data that flow into and out of thefunction.

Figure. Context-level DFD for the SafeHomesecurity function

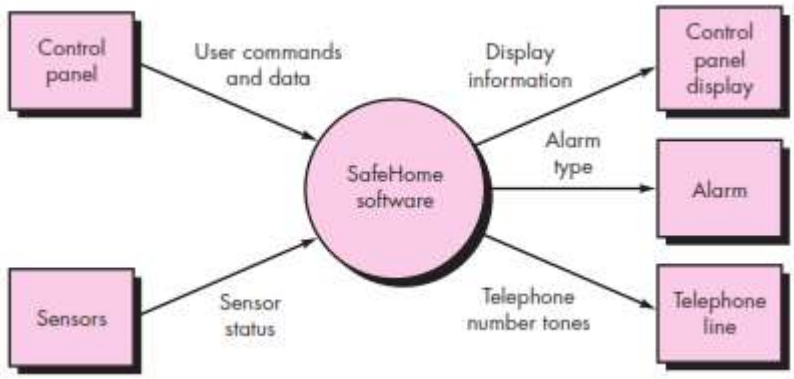


Figure. Level 0 DFD for the SafeHomesecurity function

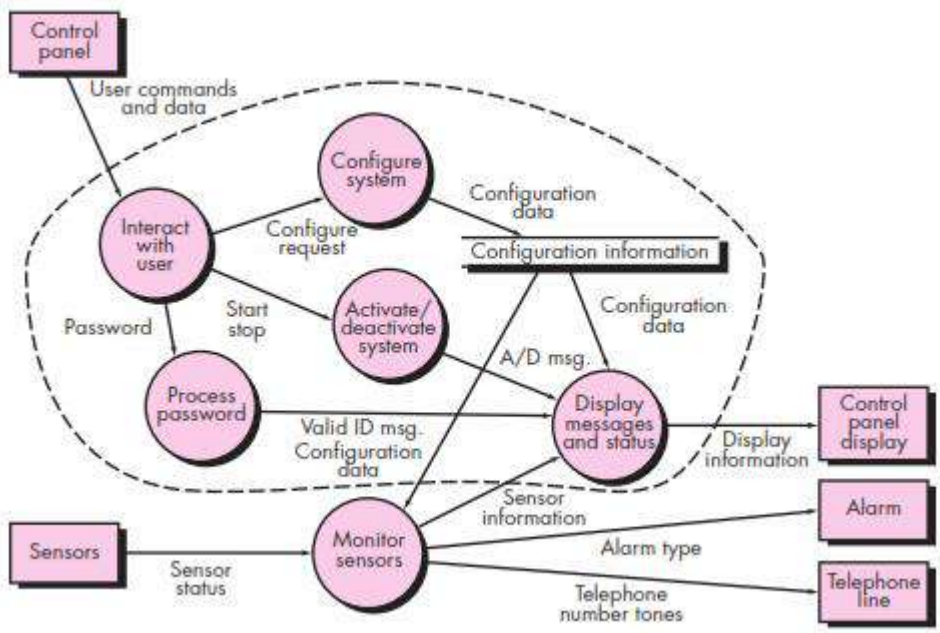


Figure. Level 2 DFD that refines the monitor sensors transform

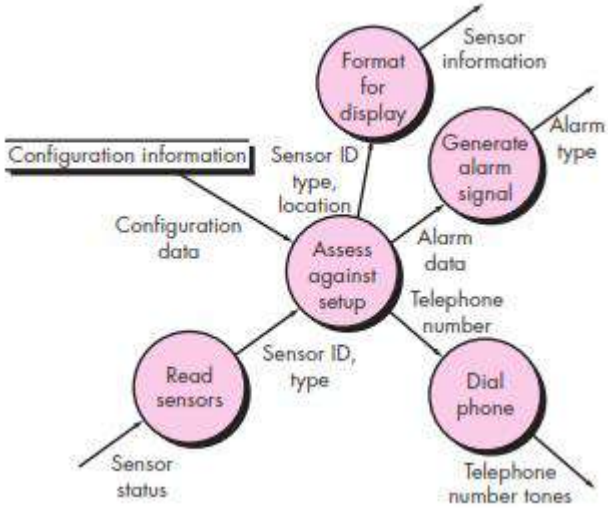
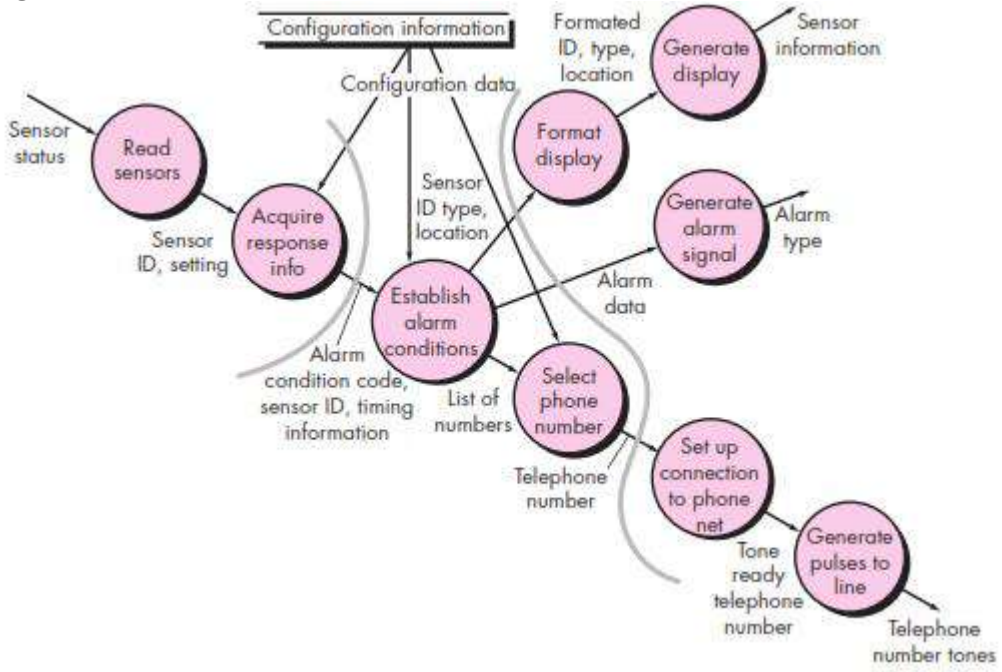


Figure. Level 3 DFD for monitor sensors with flow boundaries



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

- Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors is examined, and a level 3 data flow diagram is derived. At level 3, each transform inthe data flow diagram exhibits relatively high cohesion.
- That is, the Process implied by a transform performs a single, distinct function that can be implemented as a component in the SafeHome software. Therefore, the DFD inFigure contains sufficient detail for a "first cut" at the design of architecture for the monitor sensors subsystem, and we proceed without further refinement.

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

- Evaluating the DFD, we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoingflow 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. Incoming and outgoing flow boundaries are open to interpretation.
- That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impacton the final program structure.

Step 5. Perform "first-level factoring."

- The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top-level components perform decision making and low level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.
- When transform flow is encountered, a DFD is mapped to a specific structure(a call and return architecture) that provides control for incoming, transform, and outgoing information processing. A main controller (called monitor sensorsexecutive) 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 (e.g., a module that invokes various data transformation procedures).
• An outgoing information processing controller, called alarm output controller,coordinates production of output information.

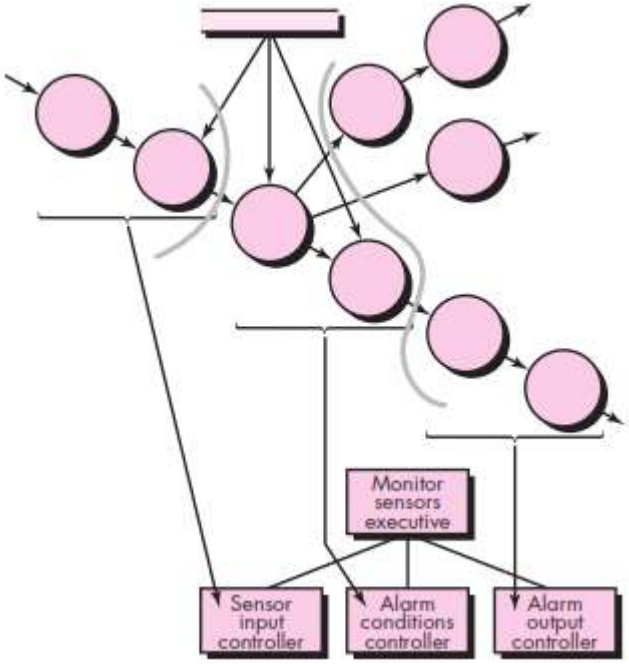


Figure. First-level factoring for monitor sensors

Step 6. Perform "second-level factoring."

- Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second level factoring is illustrated in Figure.
- Although Figure illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components. Practical considerations and measures of design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration"design.
- Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of monitor sensors subsystem software is mapped some what differently. A completed first-iteration architecture is shownin Figure .
- The components mapped in the preceding manner and shown in Figure represent an initial design of software architecture. Although components are named in a manner that implies function, a brief processing narrative (adapted fromthe process specification developed for a data transformation created duringrequirements modeling) should be written for each.

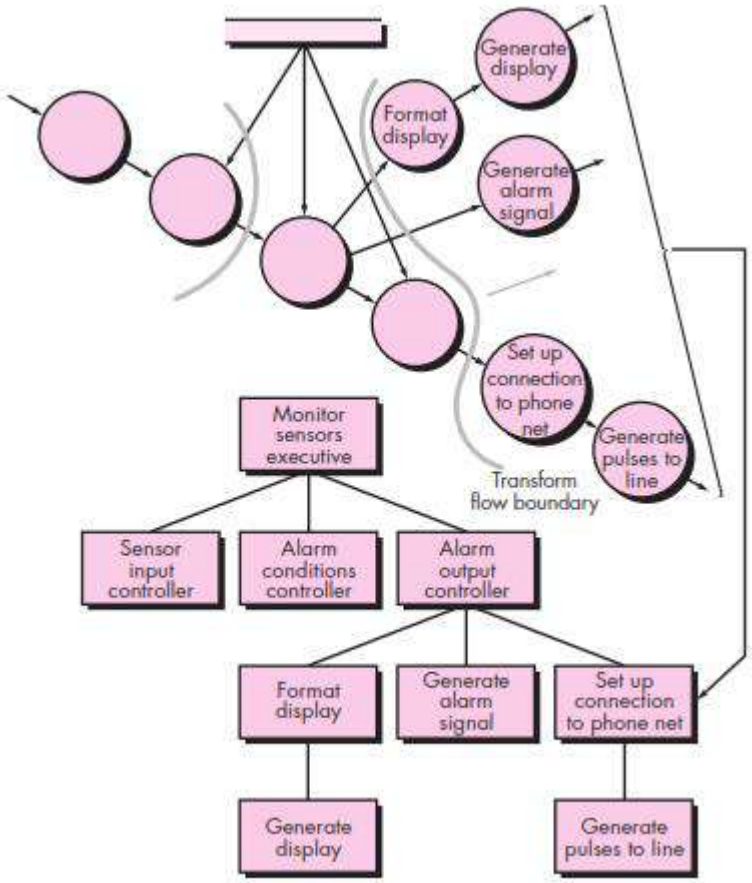


Figure. Second-level factoring for monitor sensors

- The narrative describes the component interface, internal data structures, a functional narrative, and a brief discussion of restrictions and special features (e.g., file input-output, hardware dependent characteristics, special timing requirements).

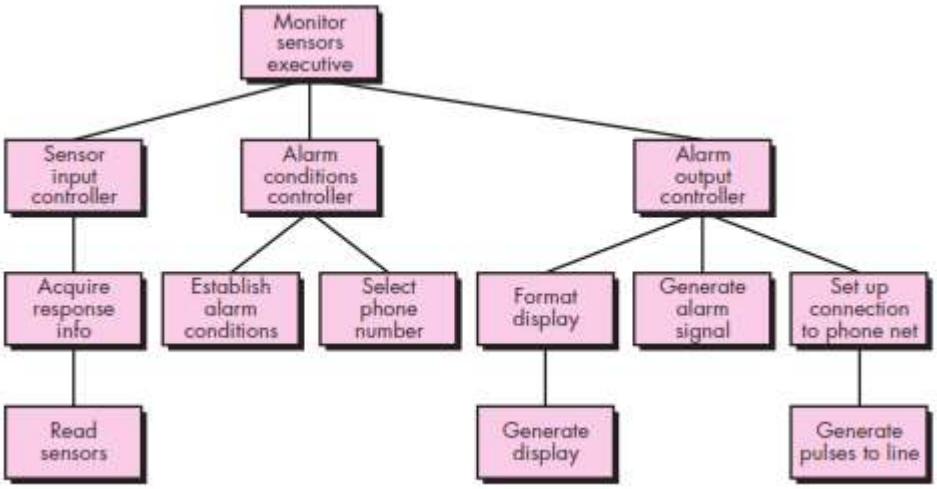


Figure. First-iteration structure for monitor sensors

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 functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion,minimal coupling, and most important, a structure that can be implemented withoutdifficulty, tested without confusion, and maintained without grief.
- Refinements are dictated by the analysis and assessment methods, as well as practical considerations and common sense.
- The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by

viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

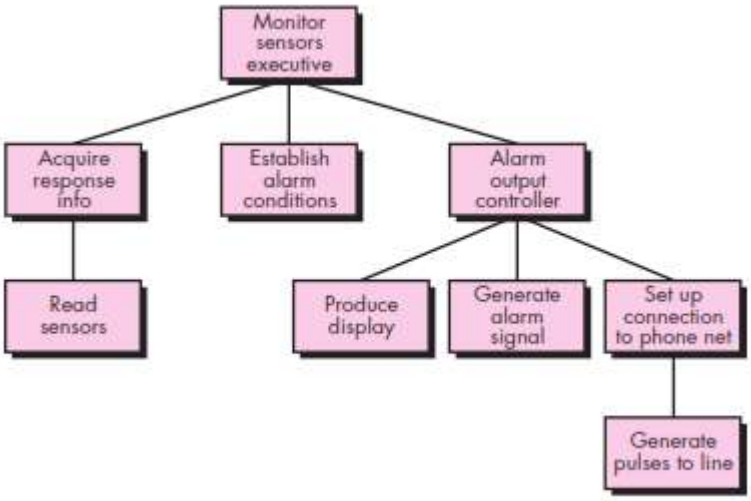


Figure. Refined program structure for monitor sensors

3.10. USER INTERFACE DESIGN

User interface design creates an effective communication medium between a human and a computer.

1. GOLDEN RULES:

1) Place the user in control.
2) Reduce the user's memory load.
3) Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

1) Place the User in Control:

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom?.

As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use.

Design principles that allow the user to maintain control are

1. Use modes judiciously (modeless)
2. Allow users to use either the keyboard or mouse (flexible)
3. Allow users to change focus (interruptible)
4. Display descriptive messages and text(Helpful)
5. Provide immediate and reversible actions, and feedback (forgiving)
6. Provide meaningful paths and exits (navigable)
7. Accommodate users with different skill levels (accessible)
8. Make the user interface transparent (facilitative)
9. Allow users to customize the interface (preferences)
10. Allow users to directly manipulate interface objects (interactive)

2) Reduce the User's Memory Load:

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall

Design principles that enable an interface to reduce the user's memory load are

1. Relieve short-term memory (remember)
2. Rely on recognition, not recall (recognition)

3. Provide visual cues (inform)
4. Provide defaults, undo, and redo (forgiving)
5. Provide interface shortcuts (frequency)
6. Promote an object-action syntax (intuitive)
7. Use real-world metaphors (transfer)
8. User progressive disclosure (context)
9. Promote visual clarity (organize)

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 rules that are maintained throughout all screen displays

(2) Input mechanisms are constrained to a limited set that is used consistently throughout the application

(3) Mechanisms for navigating from task to task are consistently defined and implemented.

Set of design principles that help make the interface consistent are:

1. Sustain the context of users' tasks (continuity)
2. Maintain consistency within and across products (experience)
3. Keep interaction results the same (expectations)
4. Provide aesthetic appeal and integrity (attitude)
5. Encourage exploration (predictable)

2. 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.

1) Interface Analysis and Design Models:

Four different models come into play when a user interface is to be analyzed and designed.

i) **User model**: Establishes the profile of the end-users of the system Based on age, gender, physical abilities, education, cultural or ethnic background, motivation, goals, and personality. A human engineer or the software engineer establishes a user model

ii) **Design model**: The software engineer creates a design model. Derived from the analysis model of the requirements. Incorporates data, architectural, interface, and procedural representations of the software.

iii) **Mental model**: The end user develops a mental image. Often called the user's system perception. Consists of the image of the system that users carry in their heads.

iv) **Implementation model:** The implementers of the system create an implementation model. Consists of the look and feel of the interface combined with all supporting information (books, videos, help files) that describe system syntax and semantics

Users can be categorized as:

i) **Novices:** No syntactic knowledge1 of the system and little semantic knowledge2 of the application or computer usage in general.

ii) **Knowledgeable, intermittent users:** Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

iii) **Knowledgeable, frequent users:** Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

2) The Process:

The analysis and design process for user interfaces is iterative and can be represented using a spiral model.

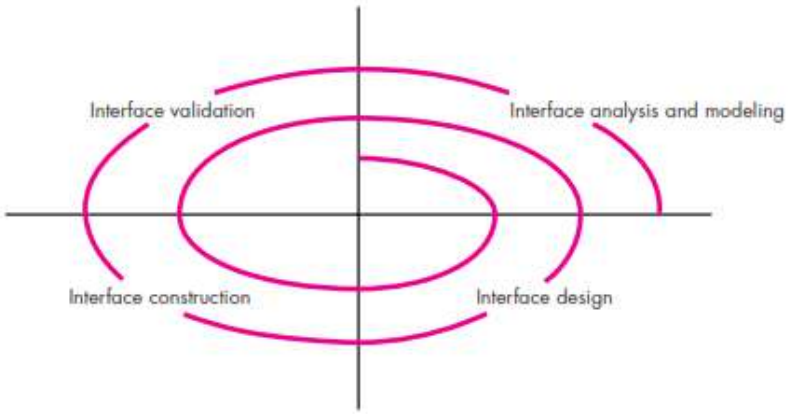


Fig. The user interface design process

Four distinct framework activities are
(1) Interface analysis and modeling
(2) Interface design
(3) Interface construction
(4) Interface validation.

The spiral implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design.

In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

(1) 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.
- ❖ For each user category, requirements are elicited. In essence, understand the system perception for each class of users.
- ❖ Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated over a number of iterative passes through the spiral.
- ❖ Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

- ❖ The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action commences.

(2) The **goal of interface design** is to define a set of interface objects, actions and their screen representations that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

(3) Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

(4) Interface validation focuses on

- ❖ The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;
- ❖ The degree to which the interface is easy to use and easy to learn
- ❖ The users' acceptance of the interface as a useful tool in their work.
- ❖ Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

3.11. INTERFACE ANALYSIS

Understand the problem before you attempt to design a solution. In the case of user interface design, 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
(4) The environment in which these tasks will be conducted.

1. User Analysis:

- The phrase "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 can be used.

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. This can be accomplished in one-on-one meetings or through focus groups.

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. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions will help you to better understand the users of a system:
- Are users trained professionals, technicians, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

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?

Use cases.
- The use case describes the manner in which an actor interacts with a system. When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task.

- In most instances, the use case is written in an informal style (a simple paragraph) in the first-person.
- Use case provides a basic description of one important work task for the computer-aided design system. From it, you can extract tasks, objects, and the overall flow of the interaction.

Task elaboration.

- Elaboration is a mechanism for refining the processing tasks that are required for software to accomplish some desired function.
- Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.

Task analysis can be applied in **two ways.**

i) An interactive computer-based system is often used to replace a manual or semi-manual activity. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform and then map these into a similar set of tasks that are implemented in the context of the user interface.

ii) Study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception.

- Regardless of the overall approach to task analysis, first define and classify tasks.

Example :

- By observing an interior designer at work, interior design comprises a number of major activities: furniture layout, fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks.

Using information contained in the use case, furniture layout can be refined into the following tasks:
(1) Draw a floor plan based on room dimensions,
(2) Place windows and doors at appropriate locations,
(3a) use furniture templates to draw scaled furniture outlines on the floor plan,
(3b) use accents templates to draw scaled accents on the floor plan,
(4) Move furniture outlines and accent outlines to get the best placement,
(5) Label all furniture and accent outlines,
(6) Draw dimensions to show location, and
(7) Draw a perspective-rendering view for the customer.

Object elaboration.

- Rather than focusing on the tasks that a user must perform, 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.
- For example, the furniture template might translate into a class called **Furniture** with attributes that might include size, shape, location, and others.
- The interior designer would select the object from the **Furniture** class, move it to a position on the floor plan (another object in this context), draw the furniture outline, and so forth.
- The tasks select, move, and draw are operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

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. This technique allows you to understand how a work process is completed when several people (and roles) are involved.

- ❖ Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients.
- ❖ Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram).
- ❖ We consider only a small part of the work process: the situation that occurs when a patient asks for a refill.
- ❖ swimlane diagram indicates the tasks and decisions for each of the three roles noted earlier. This information may have been elicited via interview or from use cases written by each actor.
- ❖ Regardless, the flow of events enables you to recognize a number of key interface characteristics:

Hierarchical representation.

- ❖ A process of elaboration occurs as you begin to analyze the interface. 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.

- ❖ To complete the task, three subtasks are defined. One of these subtasks, provide identifying information, is further elaborated in three additional sub-subtasks.

3. Analysis of Display Content:

- ❖ The user tasks identified lead to the presentation of a variety of different types of content.
- ❖ For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or videofiles).
- ❖ The analysis modeling techniques identify the output data objects that are produced by an application.

These data objects may be
(1) Generated by components in other parts of an application
(2) Acquired from data stored in a database that is accessible from the application
(3) Transmitted from systems external to the application in question.

- ❖ During this interface analysis step, the format and aesthetics of the content are considered. Among the questions that are asked and answered are:

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warnings be presented to the user?

The answers to these questions will help to establish requirements

4. Analysis of the Work Environment:

- ❖ People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people.
- ❖ If the products you design do not fit into the environment, they may be difficult or frustrating to use.
- ❖ 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.

- ❖ In addition to physical environmental factors, the workplace culture also comes into play.
- Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)?
- Will two or more people have to share information before an input can be provided?
- How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

3.12. INTERFACE DESIGN STEPS

- ❖ Once interface 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.
- ❖ Interface design is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.
- ❖ Although many different user interface design models have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.
Regardless of the sequence of design tasks, you should
(1) Always follow the golden rules
(2) Model how the interface will be implemented
(3) Consider the environment (e.g., display technology, operating system, development tools) that will be used.

1. Applying Interface Design Steps:

- ❖ The definition of interface objects and the actions that are applied to them is an important step in interface design.
- ❖ To accomplish this, user scenarios are parsed. That is, a use case is written.
- ❖ Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.
- ❖ Once the objects and actions have been defined and elaborated iteratively, they are categorized by type.
- ❖ Target, source, and application objects are identified. A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon).
- ❖ The implication of this action is to create a hard-copy report. An application object represents application-specific data that are not directly manipulated as part of screen interaction.
- ❖ **For example**, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction.
- ❖ When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed.
- ❖ Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted.
- ❖ If a real-world metaphor is appropriate for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor.
- ❖ To provide a brief illustration of the design steps noted previously, consider a user scenario for the SafeHome system (discussed in earlier chapters). A preliminary use case (written by the homeowner) for the interface follows:

Based on this use case, the following homeowner tasks, objects, and data items are identified:

- 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** (four frames per second)
- pans or zooms the **video camera**

❖ Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window with video display).

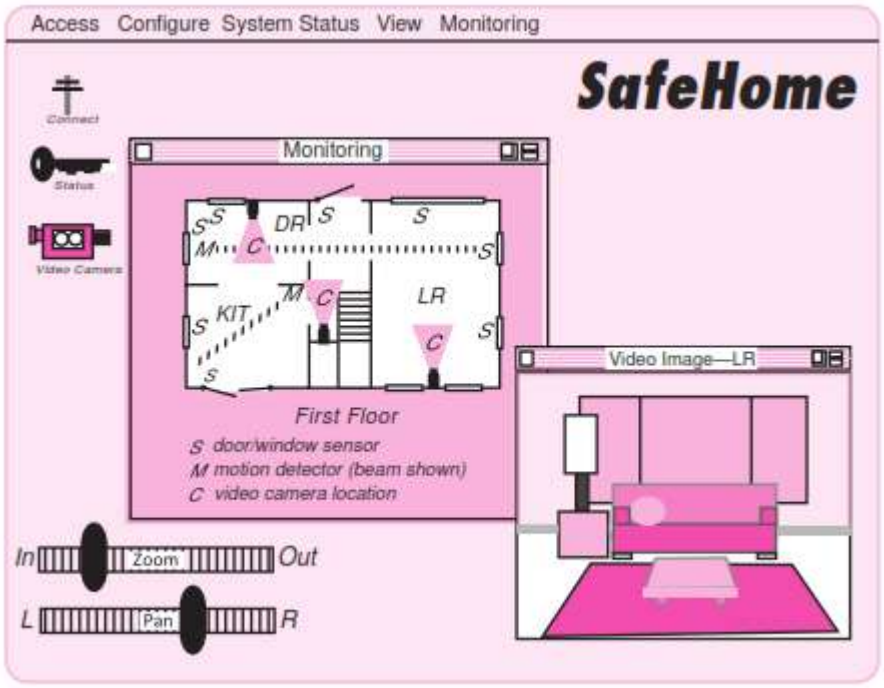


Fig. Preliminary screen layout

❖ A preliminary sketch of the screen layout for video monitoring is created . To invoke the video image, a video camera location icon, C, located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen.

❖ The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image.

❖ To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

❖ The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

2. User Interface Design Patterns:

❖ Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As I noted earlier in this book, a design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

❖ As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. There are many possible solutions to this simple problem, and a number of different patterns that might be proposed.

❖ Laakso suggests a pattern called **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be selected by picking them from the calendar. The calendar metaphor is well known to every user and provides an effective mechanism for placing a future date in context.

3. Design Issues:

Six common design issues are

i) System response time
ii) User help facilities
iii) Error information handling
iv) Command labeling

v) Application accessibility.
vi) Internationalization**.**

- ❖ Unfortunately, many designers do not address these issues until relatively late in the design process.
- ❖ Unnecessary iteration, project delays, and end-user frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

i) 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 (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.
- ❖ System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable.
- ❖ Variability refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long.
- ❖ For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "different" has occurred behind the scenes.

ii) Help facilities.

- ❖ Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of "user manuals" may be the only option.
- ❖ In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

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? Options include help for only a subset of all functions and actions or help for all functions.
- How will the user request help? Options include a help menu, a special function key, or a HELP command.
- How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.
- How will help information be structured?
- Options include a "flat" structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

iii) 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.
- ❖ There are few computer users who have not encountered an error of the form: "Application XXX has been forced to quit because an error of type 1023 has been encountered." Somewhere, an explanation for error 1023 must exist; otherwise, why would the designers have added the identification?
- ❖ Yet, the error message provides no real indication of what went wrong or where to look to get additional information. An error message presented in this manner does nothing to assuage user anxiety or to help correct the problem.

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 (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

- The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."
- The message should be "nonjudgmental." That is, the wording should never place blame on the user.

❖ Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

iv) 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.

❖ Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

v) Application accessibility.

❖ Accessibility for users who may be physically challenged is an imperative for ethical, legal, and business reasons.

❖ A variety of accessibility guidelines many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility.

❖ Others provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

vi) Internationalization.

❖ Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then
Make shift to work in other countries.

❖ 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. Localization features enable the interface to be customized for a specific market.

❖ A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues and discrete implementation issues The Unicode standard has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

3.13. COMPONENT-LEVEL DESIGN

Introduction

❖ Component-level design occurs after the first iteration of the architectural design

❖ It strives to create a design model from the analysis and architectural models
  - ✓ The translation can open the door to subtle errors that are difficult to find and correct later
  - ✓ "Effective programmers should not waste their time debugging – they should not introduce bugs to start with." Edsgar Dijkstra

❖ A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code

- ❖ The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

The Software Component

- ❖ A software component is a modular building block for computer software
  - ✓ It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- ❖ A component communicates and collaborates with
  - ✓ Other components
  - ✓ Entities outside the boundaries of the system
- ❖ Three different views of a component
  1) An object-oriented view
  2) A conventional view
  3) A process-related view

1) Object-oriented View:

- ❖ A component is viewed as a set of one or more collaborating classes
- ❖ Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
  - ✓ This also involves defining the interfaces that enable classes to communicate and collaborate
- ❖ This elaboration activity is applied to every component defined as part of the architectural design
- ❖ Once this is completed, the following steps are performed
  - ✓ Provide further elaboration of each attribute, operation, and interface
  - ✓ Specify the data structure appropriate for each attribute
  - ✓ Design the algorithmic detail required to implement the processing logic associated with each operation
  - ✓ Design the mechanisms required to implement the interface to include the messaging that occurs between objects

2) Conventional View:

- ❖ A component is viewed as a functional element (i.e., a module) of a program that incorporates
  - ✓ The processing logic
  - ✓ The internal data structures that are required to implement the processing logic
  - ✓ An interface that enables the component to be invoked and data to be passed to it
- ❖ A component serves one of the following roles
  - ✓ A control component that coordinates the invocation of all other problem domain components
  - ✓ A problem domain component that implements a complete or partial function that is required by the customer
  - ✓ An infrastructure component that is responsible for functions that support the processing required in the problem domain

- ❖ Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
  - ✓ Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
  - ✓ Control components reside near the top
  - ✓ Problem domain components and infrastructure components migrate toward the bottom
  - ✓ Functional independence is strived for between the transforms
- ❖ Once this is completed, the following steps are performed for each transform
  - ✓ Define the interface for the transform (the order, number and types of the parameters)
  - ✓ Define the data structures used internally by the transform
  - ✓ Design the algorithm used by the transform (using a stepwise refinement approach)

3) Process-related View:

- ❖ Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- ❖ As the software architecture is formulated, components are selected from the library and used to populate the architecture
- ❖ Because the components in the library have been created with reuse in mind, each contains the following:

- ✓ A complete description of their interface
- ✓ The functions they perform
- ✓ The communication and collaboration they require

3.14. 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 classes contained in the requirements model.
- The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

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. Use these principles as a guide as each software component is developed.

Component-level design principles:

❖ **Open-closed principle(OCP):**

- ✓ A module or component should be open for extension but closed for modification.
- ✓ The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component.

Example:

- Assume that the *SafeHome* security function makes use of a **Detector** class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow.
- If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP.
- One way to accomplish OCP for the **Detector** class is illustrated in Figure. The *sensor* interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the **Detector** class (component). The OCP is preserved.

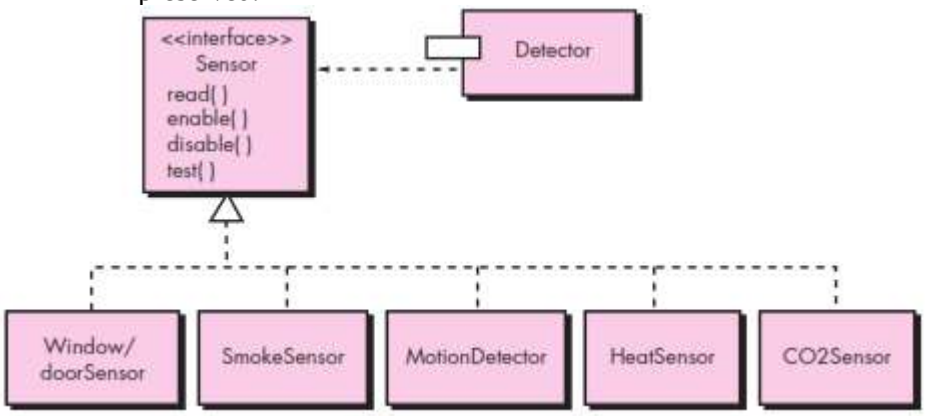


Fig. Following the OCP

❖ **Liskov substitution principle(LSP):**

- ✓ Subclasses should be substitutable for their base classes.
- ✓ A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead.

❖ **Dependency inversion principle(DIP):**

- ✓ Depend on abstractions (i.e., interfaces); do not depend on concretions.
- ✓ The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend.

❖ **Interface segregation principle(ISP):**

- ✓ Many client-specific interfaces are better than one general purpose interface

- ✓ For a server class, specialized interfaces should be created to serve major categories of clients.
- ✓ Only those operations that are relevant to a particular category of clients should be specified in the interface .

Component Packaging Principles

- ❖ **Release reuse equivalency principle(REP):**
  - ✓ The granularity of reuse is the granularity of release.
  - ✓ Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created.
- ❖ **Common closure principle(CCP)**
  - ✓ Classes that change together belong together.
  - ✓ Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change.
- ❖ **Common reuse principle(CRP):**
  - ✓ Classes that aren't reused together should not be grouped together.
  - ✓ Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded.

2. Component-Level Design Guidelines:

- ❖ **Components**
  - ✓ Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.
  - ✓ Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator).
  - ✓ Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack).

- ❖ **Interfaces.**
  - Interfaces provide important information about communication and collaboration. However, unfettered representation of interfaces tends to complicate component diagrams. Ambler recommends that

(1) Lollipop representation of an interface should be used in lieu of the more formal UML box and
Dashed arrow approach, when diagrams grow complex;
(2) For consistency, interfaces should flow from the left-hand side of the component box;
(3) Only those interfaces that are relevant to the component under consideration should be shown, Even if other interfaces are available.

- These recommendations are intended to simplify the visual nature of UML component diagrams.

- ❖ **Dependencies and inheritance in UML**
  - ✓ Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes).

Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

3. Cohesion

- ❖ Cohesion is the "single-mindedness' of a component
- ❖ It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- ❖ The objective is to keep cohesion as high as possible.

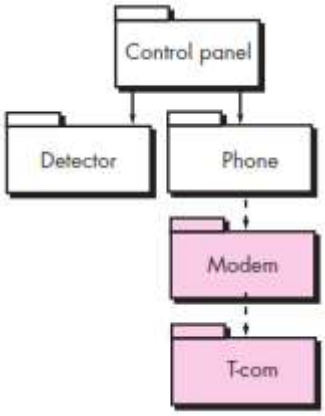


FIG. Layered cohesion

- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
    1) Functional
        - A module performs one and only one computation and then returns a result
    2) Layer
        - A higher layer component accesses the services of a lower layer component
    3) Communicational
        - All operations that access the same data are defined within one class
    4) Sequential
        - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
    5) Procedural
        - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
    6) Temporal
        - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
    7) Utility
        - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

4. Coupling:

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
    1) Data coupling
        - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
    2) Stamp coupling
        - A whole data structure or class instantiation is passed as a parameter to an operation
    3) Control coupling
        - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
        - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
    4) Common coupling
        - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
    5) Content coupling
        - One component secretly modifies data that is stored internally in another component

- **Other kinds of coupling (unranked)**
    1) Subroutine call coupling
        - When one operation is invoked it invokes another operation within side of it
    2) Type use coupling
        - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
        - If/when the type definition changes, every component that declares a variable of that data type must also change
    3) Inclusion or import coupling
        - Component A imports or includes the contents of component B
    4) External coupling
        - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

3.15. DESIGNING TRADITIONAL COMPONENTS

- Conventional design constructs emphasize the maintainability of a functional/procedural program
- The constructs are Sequence, condition, and repetition. These three constructs are fundamental to ***structured programming***—an important component-level design technique.
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- The structured constructs were proposed to limit the procedural design of software to a small number of predictable logical structures.
- Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call ***chunking.***
- Various notations depict the use of these constructs
    1) **Graphical design notation**
        - Sequence, if-then-else, selection, repetition
    2) Tabular design notation
    3) Program design language
        - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

1) Graphical Design Notation:

- A picture is worth a thousand words.
- 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.***

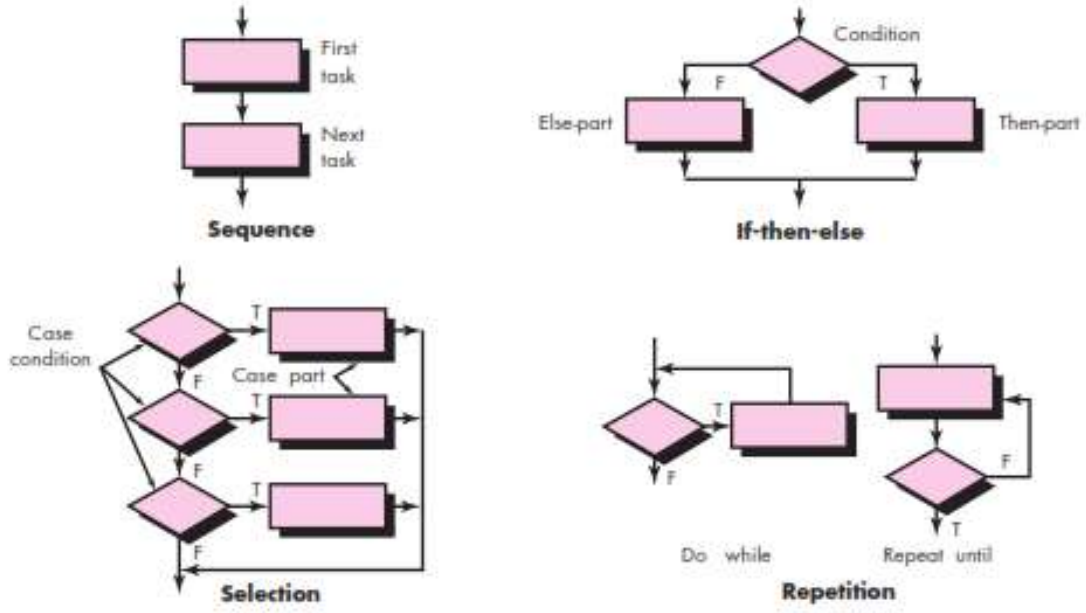


Fig. Flowchart constructs

- A flowchart, like an activity diagram, is quite simple pictorially.
- 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 (arrow) 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 shown in the figure is actually an extension of the *if-then-else.*

2) Tabular Design Notation:

- In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions.
- ***Decision tables*** provide a notation that translates actions and conditions into a tabular form. The table is difficult to misinterpret and may even be used as a machine-readable input to a table-driven algorithm.

	Rules					
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. Decision table organization

- The 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 matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. herefore, each column of the matrix may be interpreted as a ***processing rule.***

The following steps are applied to develop a decision table:

i) List all actions that can be associated with a specific procedure (or component).
ii) List all conditions (or decisions made) during execution of the procedure.
iii) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
iv) Define rules by indicating what actions occur for a set of conditions.

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).
- Narrative text (e.g., English) is embedded within a programming language-like syntax. Automated tools can be used to enhance the application of PDL.

- 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.
- It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, interprocess synchronization, and many other features.

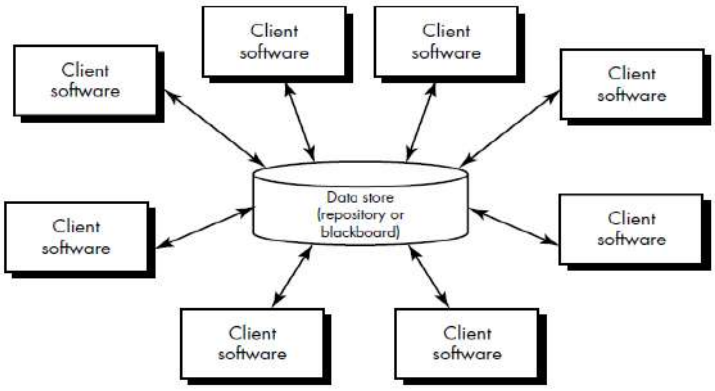
Example:

```
minmax(input)
  ARRAY a
  DO UNTIL end of input
    READ an item into a
  ENDDO
  max, min := first item of a
  DO FOR each item in a
    IF max < item THEN set max to item
    IF min > item THEN set min to item
  ENDDO
END
```

ANNA UNIVERSITY QUESTIONS AND ANSWERS
2 MARK

1. Draw diagrams to demonstrate the architectural styles. (APRIL/MAY 2015)

Data-centered architecture



2. List down the steps to be followed for User Interface design. (APRIL/MAY 2015)

1. Using information developed during interface analysis, define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

3.What are the golden rules for an interface design? (NOV/DEC 2015)

- Place the User in Control
- Reduce the User's Memory Load
- Make the Interface Consistent

4. Write a note on FURPS model of design quality. (NOV/DEC 2015) (NOV/DEC 2017)

FURPS is an acronym representing a model for classifying software quality attributes (functional and non-functional requirements):

- **F**unctionality - Capability (Size & Generality of Feature Set), Reusability (Compatibility, Interoperability, Portability), Security (Safety & Exploitability)
- **U**sability (UX) - Human Factors, Aesthetics, Consistency, Documentation, Responsiveness
- **R**eliability - Availability (Failure Frequency (Robustness/Durability/Resilience), Failure Extent & Time-Length (Recoverability/Survivability)), Predictability (Stability), Accuracy (Frequency/Severity of Error)
- **P**erformance - Speed, Efficiency, Resource Consumption (power, ram, cache, etc.), Throughput, Capacity, Scalability
- **S**upportability (Serviceability, Maintainability, Sustainability, Repair Speed) - Testability, Flexibility (Modifiability, Configurability, Adaptability, Extensibility, Modularity), Installability, Localizability

5. If a module has logical cohesion, what kind of coupling is this module likely to have? (MAY/JUNE 2016)

When a module that performs a tasks that are logically related with each other is called logically cohesive. For such module content can be suitable for coupling with other modules. The content coupling is a kind of coupling when one module makes use of data or control information maintained in other module.

6. What is the need for architectural mapping using data flow? (MAY/JUNE 2016, APRIL/MAY 2017)

It Provides a method to go from a DFD to program structure

1. The type of information flow is established
2. Flow boundaries are indicated
3. The DFD is mapped into program structure
4. Control hierarchy is defined
5. Resultant structure is refined using design measures and heuristics
6. The architectural description is refined and elaborated

7. What architectural styles are preferred for the following systems? Why? (NOV/DEC 2016)

(a) Networking - *Client server Architecture/Remote procedure call architectures*

(b) Web based systems - *N-Tier / 3-Tier Arch*itecture

(c) Banking system. - *Layered Architecture*

8. What UI design patterns are used for the following? (NOV/DEC 2016) (APRIL/MAY 2017)

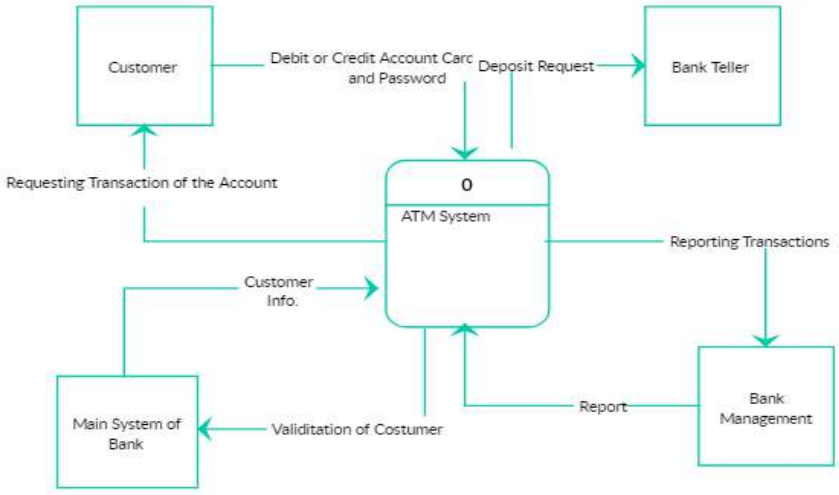
(a) **Page layout** – cards, Grid

(b) **Tables** - Alternating row color, Table filter, sort by column

(c) **Navigation through menus and web pages** - vertical dropdown menu, horizontal dropdown menu, accordion menu

(d) **Shopping cart**- product page, pricing table, coupon, shopping cart

9. Draw the context flow graph of a ATM automation system. (NOV/DEC 2017)



10. What is Inheritance? NOV/DEC 2019

Inheritance in software design model is interfacing module will be done from top (base class) to bottom (derived classes)

11. Define a component. Give example. NOV/DEC 2019

- Component-level design occurs after the first iteration of the architectural design
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

ANNA UNIVERSITY QUESTIONS
PART B

1. Explain the various coupling and cohesion methods used in Software design. **(APR/MAY 2015 and NOV/DEC 2015, APR/MAY 2017)**

2. For a Case study of your choice show the architectural and Component design.**(APR/MAY 2015)**

3. Discuss about User Interface Design of a software with an example and neat sketch **. (NOV/DEC 2015 and NOV/DEC 2017)**

4. Write short notes on the following

(i) Design heuristics

(ii) User-interface design

(iii) Component level design

(iv) Data/Class design **(APR/MAY 2016)**

5. What is modularity ? State its importance and explain coupling and cohesion. **(APR/MAY 2016)**

6. Discuss the differences between Object Oriented and Function Oriented Design. **(APR/MAY 2016)**

7. What is structured design? Illustrate the structured design process from DFD to structured chart with a case study. **(NOV/DEC 2016)**

8. Describe the golden rules for interface design**. (NOV/DEC 2016)**

9. Explain component level design with suitable examples. **(NOV/DEC 2016)**

10. What is software architecture? Describe in detail different types of software architectures with illustrations. **(APR/MAY 2017,NOV/DEC 2019) – Architectural styles**

11. Discuss about the design concepts in a software development process. **(NOV/DEC 2017)**

12. Outline the steps in designing class based components with an example**. NOV/DEC 2019**

UNIT IV- TESTING AND IMPLEMENTATION

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-Maintenance and Reengineering – BPR model – Reengineering process model-Reverse and Forward Engineering.

4.1. SOFTWARE TESTING FUNDAMENTALS:

Objective of Testing:

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. The tests must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability.

"Software testability is simply how easily a computer program can be tested."

Characteristics of testability:

1. **Operability** - "The better it works, the more efficiently it can be tested."
2. **Observability -** "What you see is what you test."
3. **Controllability -** "The better we can control the software, the more the testing can be automated and optimized."
4. **Decomposability -** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."
5. **Simplicity -** "The less there is to test, the more quickly we can test it."
6. **Stability -** "The fewer the changes, the fewer the disruptions to testing."
7. **Understandability -** "The more information we have, the smarter we will test."

Test Characteristics.

The following are attributes of a "good" test:

1) A good test has a high probability of finding an error.
2) A good test is not redundant.
3) A good test should be "best of breed"

4.2. INTERNAL AND EXTERNAL VIEWS OF TESTING:

(OR)

WHITE BOX AND BLACK BOX TESTING

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

The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing.

1. **Black-box testing (External testing):**

   Black-box testing are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

2. **White-box testing(Internal Testing):**
   White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

4.3. WHITE-BOX TESTING

White-box testing, sometimes called glass-box testing or structural testing is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.

Using white-box testing methods, you can derive test cases that

(1) Guarantee that all independent paths within a module have been exercised at least once

(2) Exercise all logical decisions on their true and false sides

(3) Execute all loops at their boundaries and within their operational bounds

(4) Exercise internal data structures to ensure their validity.

4.3.1 BASIS PATH TESTING

Basis path testing is a white-box testing technique. 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.

4.3.1.1 Flow Graph Notation

The flow graph depicts logical control flow using the notation illustrated in Figure. Each structured construct has a corresponding flow graph symbol.

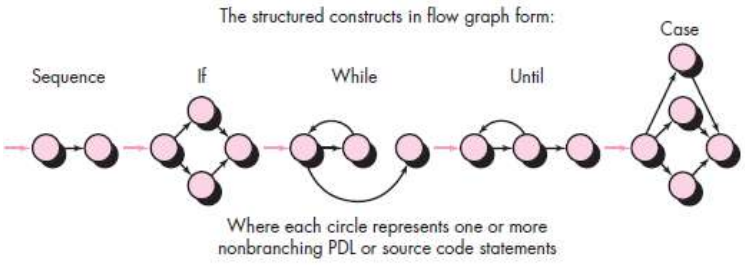
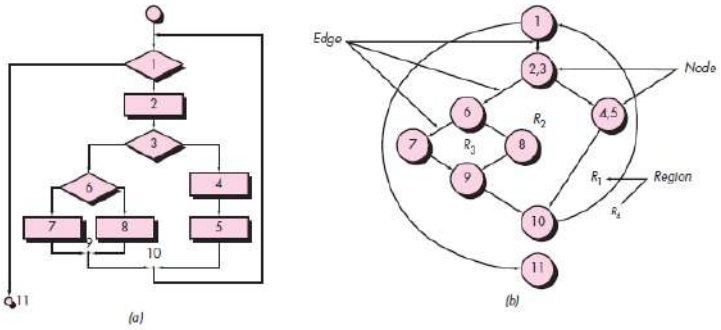


Figure. Flow graph Notation

- **Each circle, called a flow graph node**, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements. Areas bounded by edges and nodes are called regions. **When counting regions, we include the area outside the graph as a region.**



(a) Flowchart and (b) flow graph

4.3.1.2. Independent Program Paths

- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure (b) 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

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

Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity V(G) for a flow graph G is defined as

V(G)= E- N+ 2

where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as

V(G) = P+ 1

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

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

.

4.3.1.3. Deriving Test Cases

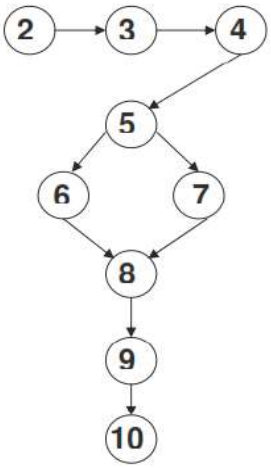
The basis path testing method can be applied to a procedural design or to source code. The following steps can be applied to derive the basis set:

1) Using the design or code as a foundation, draw a corresponding flow graph.

A flow graph is created using the symbols and construction rules .

2) Determine the cyclomatic complexity of the resultant flow graph.

The cyclomatic complexity V(G) is determined by applying the algorithms . It should be noted that V(G) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1.



Compute Cyclomatic Complexity using formulas

V(G) = e – n + 2

= 9 – 9 + 2 = 2

Therefore we have to find 2 independence paths for basis path testing

3) Prepare test cases that will force execution of each path in the basis set.

Independent path	**X**	**Y**	**Expected Result (z)**
Path 1 2-3-4-5-6-8-9-10	**10**	**5**	**5** **End program**
Path 2 2-3-4-5-7-8-9-10	**5**	**10**	**5** **End program**

4) Determine a basis set of linearly independent paths.

The value of V(G) provides the upper bound on the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify 2 paths:

Path 1-: 2-3-4-5-6-8-9-10

Path 2 -: 2-3-4-5-7-8-9-10

- Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results.

4.3.1.4 .Graph Matrices

- A **graph matrix** is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

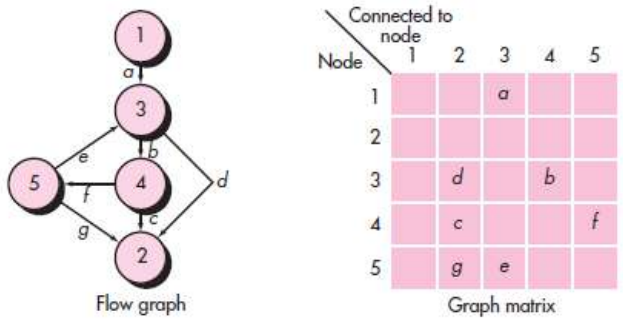


Figure. Graph matrix

- The **link weight** provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be 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.

The analysis required to design test cases can be partially or fully automated.

4.3.2 CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. The following control structure testing broadens testing coverage and improves the quality of white-box testing.

1) Condition Testing:

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E1 <relational-operator> E2

where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <,<=,=,!= (nonequality),>, or >=.

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.
- For a statement with S as its statement number,

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

(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 to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.
- A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

3) Loop Testing

- Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. 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.

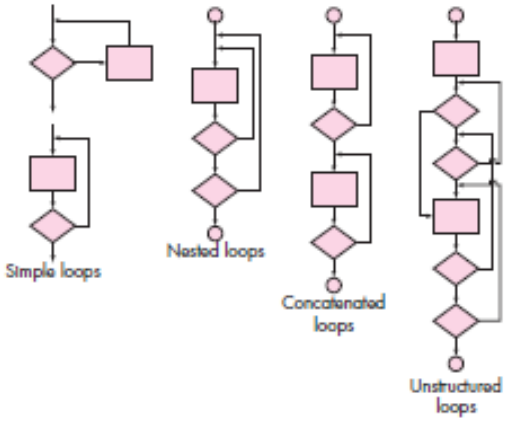


Figure. Classes of Loops

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.

- If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. an approach that will help to reduce the number of tests are:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

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 is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops.

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

4.4. BLACK-BOX TESTING

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

(1) Incorrect or missing functions
(2) Interface errors
(3) Errors in data structures or external database access
(4) Behavior or performance errors
(5) Initialization and termination errors.

By applying black-box techniques, a set of test cases can be derived that satisfy the following criteria:

(1) Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing

(2) Test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

4.4.1. Graph-Based Testing Methods

- The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another".
- Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

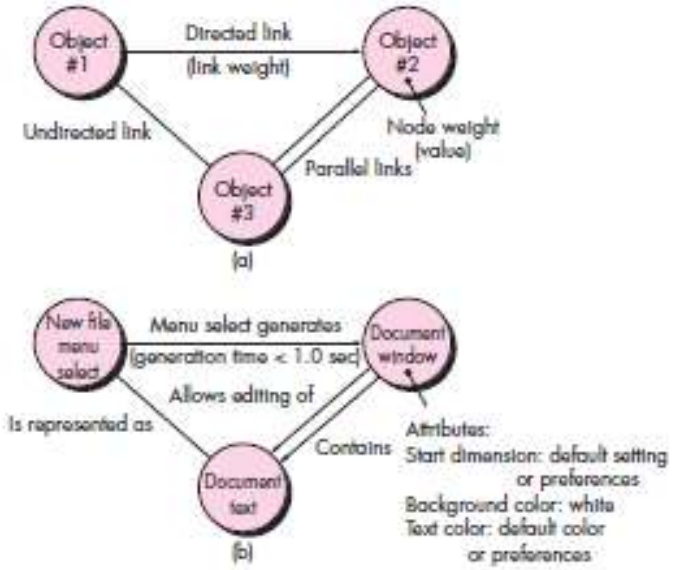


Figure. (a) Graph notation; (b) simple example

A number of behavioral testing methods that can make use of graphs are:

1) Transaction flow modeling.
2) Finite state modeling.
3) Data flow modeling.
4) Timing modeling.

4.4.2 .Equivalence Partitioning

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.
- Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present.
- An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition 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.

Example#1:

For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three quivalence classes:

The set of negative integers,the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each ofthe three equivalence classes and a possible test set can be: {-5,500,6000}.

Example#2:

Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.

The equivalence classes are the following:

•Parallel lines (m1=m2, c1≠c2)

•Intersecting lines (m1≠m2)

•Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7) , (10, 10) (10, 10) are obtained.

4.4.3. Boundary Value Analysis

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

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

1) If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.

2) If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3) Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4) If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

- Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

4.4.4. Orthogonal Array Testing

- . Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component.
- When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a "balancing property".
- **Detect all double mode faults.**
  If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault. Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.
- **Multimode faults.**
  Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multimode faults are also detected by these tests.

4.5. REGRESSION TESTING:

- Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly.
- 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.
- In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated capture/playback tools. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

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

- As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.
- It is impractical and inefficient to reexecute every test for every program function once a change has occurred.

4.6. UNIT TESTING:

- Unit testing focuses verification effort on the smallest unit of software design—the software component or module.
- The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

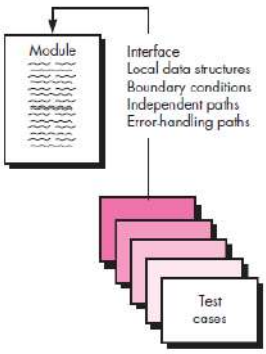


Figure. Unit test

Unit-test considerations:

- The module interface is tested to ensure that information properly flows into and out of the program.
- Local data structures are examined to ensure that integrity is maintained.
- All independent paths are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- All error handling paths should be tested.

Unit-test procedures:

- The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.
- Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.

- In most applications a **driver i**s nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (invoked by) the component to be tested.
- A **stub** or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Drivers and stubs represent testing "overhead." That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low.
- Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).
- Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

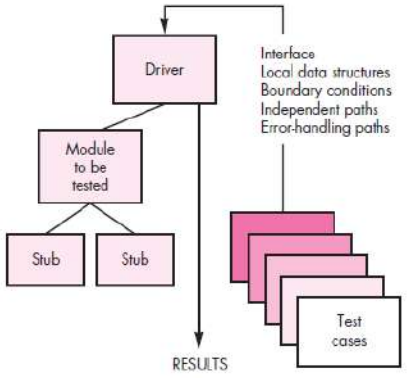


Figure. Unit-test environment

4.7. INTEGRATION TESTING:

- **Integration testing** is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.
- There is often a tendency to attempt **non incremental integration**; that is, to construct the program using a **"big bang" approaches.** All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.
- In **Incremental integration**, the program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied..

4.7.1.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.

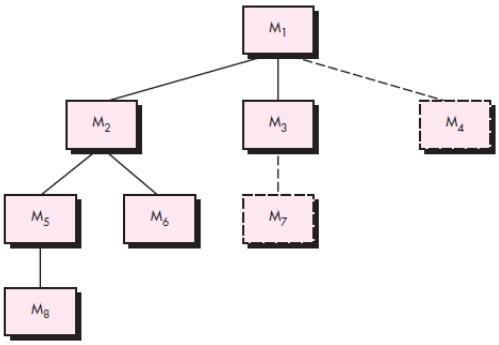


Figure. Top-down integration

- Referring to Figure, **depth-first integration** integrates all components on a major control path of the program structure.
- **For example**, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built.
- **Breadth-first integration** incorporates all components directly subordinate at each level, moving across the structure horizontally.
- From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

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

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.

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.

4.7.2.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 subfunction.

2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

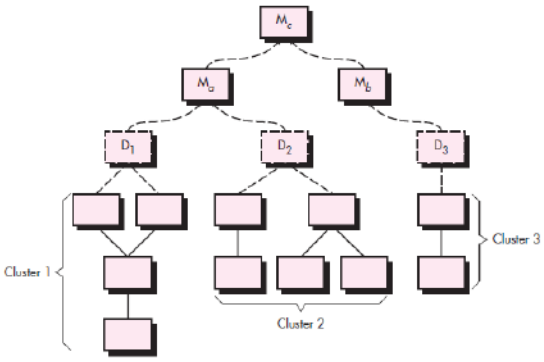


Figure. Bottom-up Integration

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

4.7.3. Smoke Testing:

Smoke testing is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

Smoke-testing 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 daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress.

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

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

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

4.8 . VALIDATION TESTING:

- Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.
- Validation succeeds when software functions in a manner that can be reasonably expected by the customer.

1. Validation-Test Criteria:

- Software validation is achieved through a series of tests with requirements. 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.

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.

2. 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.

3. Alpha and Beta Testing:

It is virtually impossible for a software developer to how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.

Conducted by the end user rather than software engineers, an acceptance test can range from an informal “test drive” to a planned and systematically executed series of tests.

Acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

Alpha Test:

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 Test:

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.

4.9. SYSTEM TESTING:

- Software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted.
- These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.
- A classic system-testing problem is "finger pointing." This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem.

Rather than indulging in such nonsense, you should anticipate potential interfacing problems and (1) Design error-handling paths that test all information coming from other elements of the system,

(2) conduct a series of tests that simulate bad data or other potential errors at the software interface,

(3) Record the results of tests to use as "evidence" if finger pointing does occur, and

(4) Participate in planning and design of system tests to ensure that software is adequately tested.

Types of system tests are

1) Recovery Testing
2) Security Testing
3) Stress Testing
4) Performance Testing
5) Deployment Testing

1) Recovery Testing:

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

If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.

If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

2) Security Testing:

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

"The system's security must be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester may attempt to acquire passwords through external clerical means; may attack the system, thereby denying service to others; may purposely cause

system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

3) Stress Testing:

Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

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

For example,

(1) Special tests may be designed that generate 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. Essentially, the tester attempts to break the program.

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

4) Performance Testing:

Performance testing is designed to test the run-time performance of software within the context of an integrated system.

Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box 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.

5) Deployment Testing:

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

4.10. DEBUGGING:

Software testing is a process that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

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.

1. The Debugging Process:

The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.

In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes:

(1) The cause will be found and corrected or

(2) The cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

Few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components exacerbate this situation.

2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).

4. The symptom may be caused by human error that is not easily traced.

5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

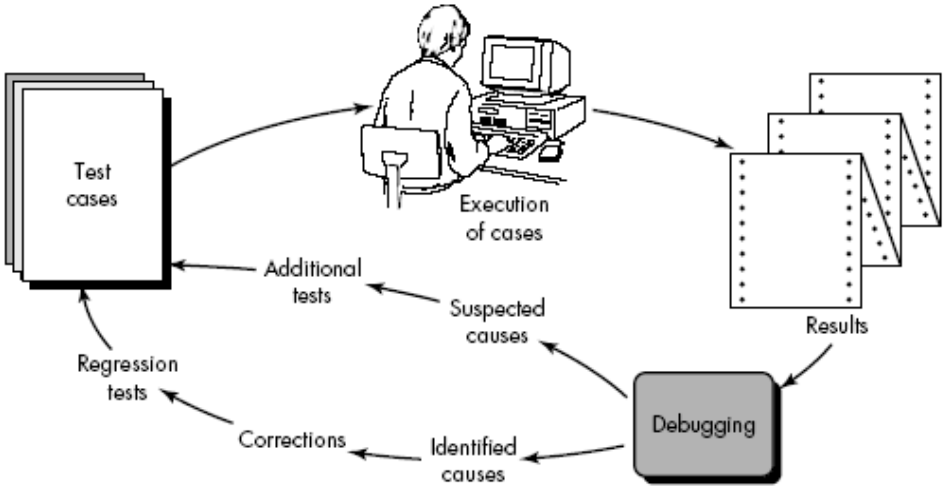


Figure.The debugging process

During debugging, you'll encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g., the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure forces some software developers to fix one error and at the same time introduce two more.

2 .Psychological Considerations:

Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake.

Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately . . . corrected.

3. Debugging Strategies:

Regardless of the approach that is taken, debugging has one overriding objective— to find and correct the cause of a software error or defect. The objective is realized by a combination of systematic evaluation, intuition, and luck.

In general, three debugging strategies have been proposed:

(1) Brute force

(2) Backtracking

(3) Cause elimination.

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

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.

Using a **"let the computer find the error"**, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements.

Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time.

2) Backtracking:

Backtracking is a fairly common debugging approach that can be used successfully in **small programs**.

Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

3) Cause elimination:

The third approach to debugging—**cause elimination—**is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.

Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each

Automated debugging.

Each of these debugging approaches can be supplemented with debugging tools that can provide you with semiautomated support as debugging strategies are attempted.

Integrated development environments (IDEs) provide a way to capture some of the language specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation."

4. Correcting the Error:

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good.

Three simple questions that you should ask before making the "correction" that removes the cause of a bug:

1. Is the cause of the bug reproduced in another part of the program?
2. What "next bug" might be introduced by the fix I'm about to make?
3. What could we have done to prevent this bug in the first place?

4.11. SOFTWARE IMPLEMENTATION TECHNIQUES:

4.11.1 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.

4.11.1.1. Coding standards

"Establish programming conventions before you begin programming. It's nearly impossible to change code to match them later."

The use of coding conventions is particularly important when a project involves more than one programmer. It is much easier for a programmer to read code written by someone else if all code follows the same conventions.

1. **Commenting**

Due to time restrictions or enthusiastic programmers who want immediate results for their code, commenting of code often takes a back seat.

Programmers working as a team have found it better to leave comments behind.However, some commenting can decrease the cost of knowledge transfer between developers working on the same module.

In the early days of computing, one commenting practice was to leave a brief description of the following:

1. Name of the module.
2. Purpose of the Module.
3. Description of the Module (In brief).
4. Original Author
5. Modifications
6. Authors who modified code with a description on why it was modified.

2. **Naming conventions**

Use of proper naming conventions is considered good practice. Sometimes programmers tend to use X1, Y1, etc. as variables and forget to replace them with meaningful ones, causing confusion.

In order to prevent this waste of time, it is usually considered good practice to use descriptive names in the code since we deal with real data.

Example: A variable for taking in weight as a parameter for a truck can be named TrkWeight or TruckWeight, with TruckWeight being the more preferable one, since it is instantly recognisable.

3. **Keep the code simple**

The code that a programmer writes should be simple. Complicated logic for achieving a simple thing should be kept to a minimum since the code might be modified by another

programmer in the future. The logic one programmer implemented may not make perfect sense to another. So, always keep the code as simple as possible.

4. **Portability**

Program code should **never ever** contain "hard-coded", values referring to environmental parameters, such as absolute file paths, file names, user names, host names, IP addresses, URLs, UDP/TCP ports.

4.11.1.2.Code development

Code building

A best practice for building code involves daily builds and testing, or better still continuous integration, or even continuous delivery.

Testing

Testing is an integral part of software development that needs to be planned. It is also important that testing is done proactively; meaning that test cases are planned before coding starts, and test cases are developed while the application is being designed and coded.

Debugging the code and correcting errors

Programmers tend to write the complete code and then begin debugging and checking for errors. Though this approach can save time in smaller projects, bigger and complex ones tend to have too many variables and functions that need attention.

4.11.2 Refactoring

Refactoring is a technique to keep the code cleaner, simpler, extendable, reusable and maintainable.

Refactoring leads to constant improvement in software quality while providing reusable, modular and service oriented components.

It is a disciplined and controlled technique for improving the software code by changing the internal structure of the code without affecting the functionalities.

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.
- **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 reaggregation and consolidation. Architecture driven refactoring is targeted to achieve certain business objectives where existing practices fails to deliver those objective.

4.11.2.1 Why Refactoring needed?

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

- **Maintainability** – Code is not easily maintainable
- **Extendibility** – Extending / adding new features in the application are not possible or very expensive.

Refactoring is needed due to various reasons e.g.

1. **Lack of Modularity** – existing feature of one application can't be used in another application due to its tightly coupling with the application components
2. **Lack of reusable components** – There are instances of code duplicity and potential reusable components dependency on application code.
3. **Lack of pluggable components** – existing components are not easily replaceable due to its application code tightly coupling with the components.
4. **Service oriented architecture -** Scope for SOA components where each component can work as a service and reusable
5. **Code redundancy -** Application has lots of dead code and duplicate code
6. **Lack of layered architecture** – Any change in one layer causing changes in all other layers
7. **Poor coding style** – Coding standards has not been followed properly – it includes improper names to object/methods, accessing the fields without getter/setters
8. **Illogical methods composition** – Illogical grouping of methods in one class.
9. **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.
10. **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.

4.11.2.2 Steps for Refactoring

1. Writing unit test cases – Test cases should be written to test the application behavior and ensure that it is unchanged after every cycle of refactoring.
2. Identifying the task for refactoring –
    a. Find the problem –what is the problem?
    b. Evaluate / Analyze the problem
3. Design solution – Find out what will be the resultant code after refactoring of the code.
4. Modify the code – Refactor the code without changing the outer behavior of the code .
5. Test refactored code - repeat the refactoring in a different way.
6. Repeat above cycle until the current code moves to the target state.

4.11.2.3 Key benefits from refactoring:

1. Improves software expendability
2. Reduces code maintenance cost
3. Provides standarised code
4. Architecture improvement without impacting software behavior
5. Provides more readable and modular code
6. Refactored modular component – increase potential reusability

4.12. MAINTENANCE AND REENGINEERING :

- Software maintenance is an activity in which program is modified after it has been put into use.
- In software maintenance usually it is not preferred to apply major software changes to system's architecture.
- Maintenance is a process in which changes are implemented by either modifying the existinmg system's architecture or by adding new components to the system.

Need for Maintenance:

- The system changes and hence maintenance must be performed in order to:
    a. Correct faults
    b. Improve design
    c. Implement enhancement
    d. Interface with other systems
    e. Adaption of environment
    f. Migrate legacy software

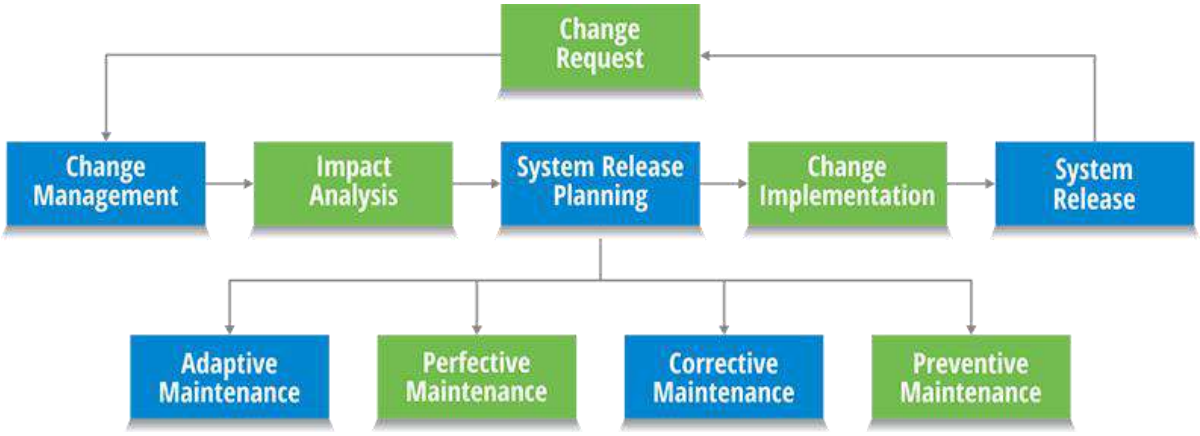
Types of Software Maintenance:

1. Adaptive – Modifications in system to keep it compatible with changing business and technical environment.
2. Perfective – Fine tuning of all elements, functionalities and abilities to improve system operations and perfectness.
3. Corrective – Detecting errors in the existing solution and correcting them to make it works more efficiently.
4. Preventive – Preventive software maintenance services help in preventing the system from any upcoming vulnerabilities.

Software Maintenance Process:

The software evolution process is dependent upon the type of software being maintained.
The software maintenance process can be as shown below

1. In the maintenance process initially the **request** for change is made.
2. **Change management:** In this phase the status of the entire change request is identified, described.
3. **Impact analysis**: following activities are performed in this phase:
    i. Identify all systems and system products affected by a change request.
    ii. Make an estimate of the resources needed to effect the change.
    iii. Analyze the benefits of the change



4. **System release planning:** in this phase the schedule and contents of software release is planned. The changes can be to all types of software maintenance.
5. **Change implementation:** The implementation of changes can be done by fact designing the changes, then coding for these changes and finally testing the changes. Preferably the regression testing must be performed while testing the changes.
6. **System release**: During the software relase i) Documentation ii) Software iii) Training iv) Hardware changes v) Data conversion should be described.

Factors affecting maintenance costs:

1. **Module Independence** – This is the ability to modify one part of the system.
2. **Programming Language** – For the higher level of the language, the maintenance is cheaper.
3. **Programming Style** – The way in which a program is written makes difference in the cost.
4. **Program Validation and Testing** – The more time and effort spent on design validation and program testing, the fewer errors and the less the need for corrective maintenance.
5. **Quality of Prof\gram Documentation** – The better the documentation, the easier it is to maintain.
6. **Configuration Management Techniques -** Keeping track of all the system documents and ensuring they are consistent is a major cost of maintenance.
7. **Age of the system –** Older systems are difficult to maintain.

Issues in Software Maintenance:

1. **Technical –** This is a key issue in software maintenance. Technical maintenance is based on following factors such as limited understanding of system, testing, impact analysis and maintainability.
2. **Management –** Management issue includes organizational issues, staffing problem, problem issue, organizational structure, outsouring.
3. **Cost Estimation -** This is one of the major issues in software maintenance. It is based on cost, experience of projects.
4. **Software maintenance measurement -** The software measurement factors such as size effort, schedule, quality, understandability, resource utilization, design complexity, reliability and fault type distribution.

4.13. BUSINESS PROCESS REENGINEERING:

Definition: The Business Process Re-engineering (BPR) is the implementation of radical change in the business process to achieve breakthrough results.

- A business process is "a set of logically related tasks performed to achieve a defined business outcome". Within the business process, people, equipment, material resources, and business procedures are combined to produce a specified result.
- Examples of business processes include designing a new product, purchasing services and supplies, hiring a new employee, and paying suppliers.

4.13.1 BPR Model:

Business process reengineering is iterative. Business goals and the processes that achieve them must be adapted to a changing business environment

The model defines six activities:

Business definition. Business goals are identified within the context of four key drivers: cost reduction, time reduction, quality improvement, and personnel development and empowerment. Goals may be defined at the business level or for a specific component of the business.

Process identification. Processes that are critical to achieving the goals defined in the business definition are identified. They may then be ranked by importance, by need for change, or in any other way that is appropriate for the reengineering activity.

Process evaluation. The existing process is thoroughly analyzed and measured. Process tasks are identified; the costs and time consumed by process tasks are noted; and quality/performance problems are isolated.

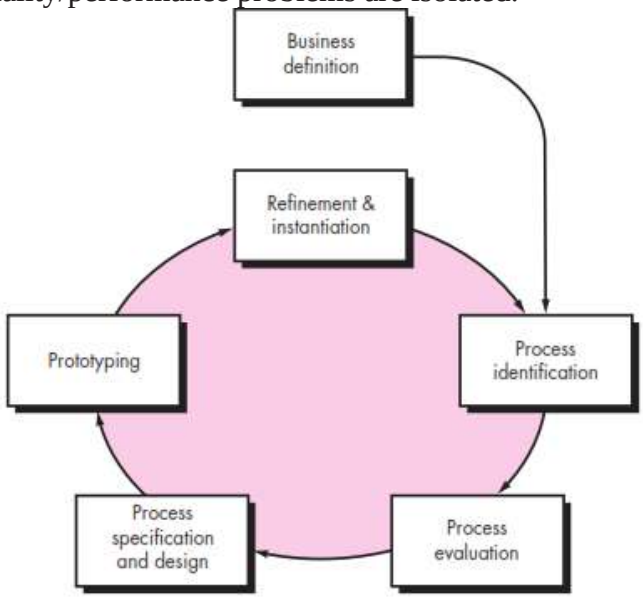


Fig: The BPR Model

Process specification and design. Based on information obtained during the first three BPR activities, use cases are prepared for each process that is to be redesigned. Within the context of BPR, use cases identify a scenario that delivers some outcome to a customer. With the use case as the specification of the process, a new set of tasks are designed for the process.

Prototyping. A redesigned business process must be prototyped before it is fully integrated into the business. This activity "tests" the process so that refinements can be made.

Refinement and instantiation. Based on feedback from the prototype, the business process is refined and then instantiated within a business system.

These BPR activities are sometimes used in conjunction with workflow analysis tools. The intent of these tools is to build a model of existing workflow in an effort to better analyze existing processes.

4.13.2 Reengineering process model:

- Software reengineering is a process of modifying the system for maintenance purpose.
- Reengineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software reengineering. A workable strategy is encompassed in a reengineering process model.

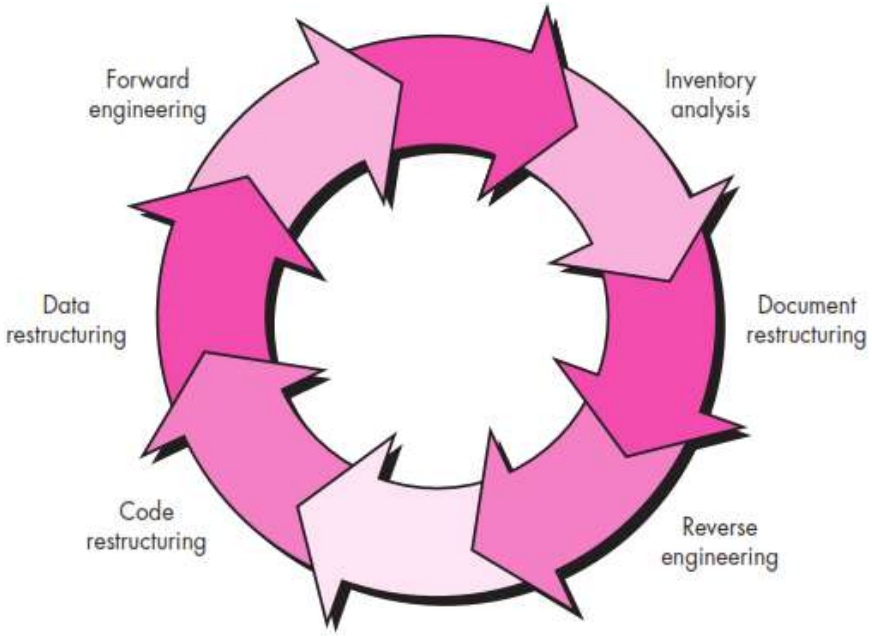


FIGURE: A software reengineering process model

For any particular cycle, the process can terminate after any one of these activities.

Inventory analysis:

- Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description of every active application.
- Resources can then be allocated to candidate applications for reengineering work. It is important to note that the inventory should be revisited on a regular cycle.

Document restructuring:

Weak documentation is the trademark of many legacy systems. But what can you do about it? What are your options?

1. *Creating documentation is far too time consuming.* If the system works, you may choose to live with what you have. In some cases, this is the correct approach. It is not possible to re-create documentation for hundreds of computer programs.
2. *Documentation must be updated, but your organization has limited resources.* You'll use a "document when touched" approach. It may not be necessary to fully redocument an application.
3. *The system is business critical and must be fully redocumented.* Even in this case, an intelligent approach is to pare documentation to an essential minimum.

Each of these options is viable. Your software organization must choose the one that is most appropriate for each case.

Reverse engineering:

- Reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code.
- Reverse engineering is a process of *design recovery*.
- Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Code restructuring:

- The most common type of reengineering is *code restructuring.*
- The source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted and code is then restructured or even rewritten in a more modern programming language.
- The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.

Data restructuring:

- A program with weak data architecture will be difficult to adapt and enhance. In most cases, data restructuring begins with a reverse engineering activity.
- Current data architecture is dissected, and necessary data models are defined.
- Data objects and attributes are identified, and existing data structures are reviewed for quality. When data structure is weak then the data are reengineered.

Forward engineering:

- Forward engineering not only recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.
- In most cases, reengineered software reimplements the function of the existing system and also adds new functions and/or improves overall performance.

4.14 REVERSE ENGINEERING

Reverse engineering is a process of *design recovery*. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

There are three important issues in reverse engineering:

1. **Abstraction level** *:*

- The ***abstraction level*** of a reverse engineering process and the tools used to refers to the sophistication of the design information that can be extracted from source code.
- The abstraction level should be as high as possible.
- As the abstraction level increases, you are provided with information that will allow easier understanding of the program.

2. Completeness Level:

- The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases.
- For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple architectural design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.
- Completeness improves in direct proportion to the amount of analysis performed by the person doing reverse engineering. *Interactivity* refers to the degree to which the human is "integrated" with automated tools to create an effective reverse engineering process.

3. Directionality level:

If the *directionality* of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.

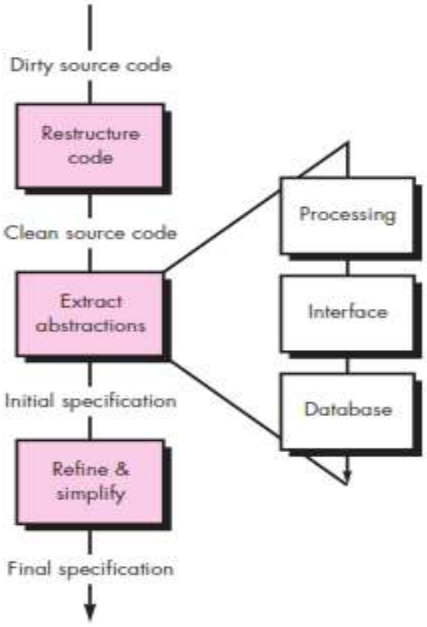


Figure - The reverse engineering process

- The reverse engineering process is represented in Figure. Before reverse engineering activities can commence, unstructured ("dirty") ource code is restructured so that it contains only the structured programming constructs.
- Dirty source code Restructure code Clean source code Extract abstractions Initial specification Refine & simplify Final specification This makes the source code easier to read and provides the basis for all the subsequent reverse engineering activities.
- The core of reverse engineering is an activity called *extract abstractions*. You must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

Difference between Software and Reverse Engineering

S. No	Software Engineering	Reverse Engineering
1	Software Engineering is a discipline in which theories, methods and tools are applied to develop a professional software product.	Reverse engineering is a process in which the dirty or unstructured code is taken processed and it is restructured.
2	Initially only user requirements are available for software engineering process.	A dirty or unstructured code is available initially
3	This process starts by understanding user requirements	This process starts by understanding the existing unstructured code
4	The software engineering is constructed using requirement gathering, analysis, design, implantation and testing	Thr reverse engineering is constructed using restructuring the code, cleaning it, by extracting the abstractions. After refinement and simplification of the code final code gets ready
5	It is simple and straightforward approach	It is complex because cleaning the dirty or unstructured code requires more efforts
6	Documentation or specification of the product is useful to the end-user	Documentation or specification of the product is useful to the developer.

Difference between Reverse Engineering and Re-Engineering

Reverse Engineering	Re-Engineering
Reverse Engineering is a process of finding out how a product works from already created software system	Re-Engineering is to observe the software system and build it again for better use.
In Reverse Engineering, the source code is re-created from the complied code	In Re-Engineering, new piece of code with similar or better functionality than the existing one is created.
Reverse Engineering is carried out for trying to understand inner working of the artifact with availability of any documents.	Re-Engineering is carried out for designing something again. Many time from scratch.

4.15 FORWARD ENGINEERING

If the poorly designed and implemented code is to be modified then following alternatives can be adopted:

1. Make lot of modifications to implement the necessary changes.
2. Understand inner workings of the program in order to make the necessary modifications.
3. Redesign, recode and test small modules of software that require modifications.
4. Completely redesign, recode and test the entire program using re-engineering tool.

Definition: Forward Engineering is a process that makes use of software engineering principles, concepts and methods to re-create an existing application. This re-developed program extends the capabilities of old programs.

Forward Engineering for Object Oriented Architecture:

Forward engineering is a process of re-engineering conventional software into the object oriented implementation.

Following are the steps that can be applied for forward engineering the conventional software:

1. Existing software is reverse engineered in order to create data, functional, and behavioral models.
2. If existing system extends the functionality of original application then use cases can be created.
3. The data models created in this process are used to create the base for classes.
4. Class Hierarchies, object-relationship models, object behavioral models and subsystems are defined.

During this forward engineering process, algorithms and data structures are reused from existing conventional application.

Difference between Forward and Reverse Engineering:

- Forward engineering is a process of constructing a system for specific purpose.
- Reverse engineering is a process of de-constructing a system in order to extend the functionalities or in order to understand the working of the system.

ANNA UNIVERSITY QUESTIONS AND ANSWERS
PART A

1. **What is the need for regression testing? (APR/MAY 2015)**

Regression testing may be conducted to ensure that new errors have not been introduced. Regression Testing is required when there is a

- Change in requirements and code is modified according to the requirement
- New feature is added to the software
- Defect fixing
- Performance issue fix

2. **Write the best practices for "CODING". (APR/MAY 2015) (NOV/DEC 2015)**
    1. Know what the code block must perform
    2. Indicate a brief description of what a variable is for (reference to commenting)
    3. Correct errors as they occur.
    4. Keep your code simple
    5. Maintain naming conventions which are uniform throughout.
3. **How will you test a simple loop? (NOV/DEC 2015)**

    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.
4. **How can refactoring be made more effective? (APR/MAY 2016)**

There are two general categories of benefits to the activity of refactoring.

1. **Maintainability.** It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp. This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods.

2. **Extensibility**. It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.

5. **Why does software fail after it has passed from acceptance testing? (APR/MAY 2016)**
   - During acceptance testing, the random input is used for testing. This may lead to the situation that some input values that may cause failure go unhandled. The practical problem with acceptance testing is that it is time consuming. Hence in order to keep testing cost low, there is restricted number of test cases.
6. **What methods are used for breaking very long expression and statements? (NOV/DEC 2016)**

Statement testing can be improved if we allow program graphs to have nodes made up of statement fragments rather than complete statements. For example, take the following line of **pseudo code:**

If x > y then x else y

With average statement testing, this statement would be placed into a single node, and so only one of the predicates would be tested. However, if we allow for statement fragments, then we would get the following pseudo code:

If x > y then

x

Else y

With each of these statements now separate, they will each be placed in a different node. As a result, statement testing will go through each node with the result that we also achieve predicate outcome coverage.

7. **What is the difference between verification and validation? Which types of testing address verification? Which types of testing address validation? (NOV/DEC 2016) (APR/MAY 2017) (NOV/DEC 2017)**

Verification	Validation
Verification testing comprise of various activities that ensure software correctly implements the specific function	Validation refers to set of activities that ensure that the software that has been built is traceable to customer requirements
The performance testing is testing address verification	The acceptance testing is testing address validation.

8. **What is smoke testing**? **(APR/MAY 2017)**

   Smoke testing is the initial testing process exercised to check whether the software under test is ready/stable for further testing

9. **Mention the purpose of stub and Driver used for testing. (NOV/DEC 2017)**

- The Drivers is a program that accepts the test data nad prints the relevant results.
- The stub is a subprogram that uses the module interfaces and performs the minimal data manipulation if required.

10. **What are the testing principles the software engineer must apply while performing the software testing? (MAY 18)**
    1. All tests must be traceable to customer requirements
    2. Tests should be planned long before testing begins
    3. Testing should begin in small and progress towards testing in large.
    4. Exhaustive testing is not possible.
    5. Testing should be done independent third party.
11. **Identify the type of maintenance for each of the following: (MAY 18)**
    a) Correcting the software faults
    b) Adapting the change in environment

    Ans: 1. Corrective Maintenance 2. Adaptive Maintenance
12. **What is test case? NOV/DEC 2019**

    A **TEST CASE** is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.
13. **Outline the needs of system testing. NOV/DEC 2019**

    A classic system-testing problem is "finger pointing." This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem.
14. **List the levels of testing. APR/MAY 2019**

    The developed software should be tested in the following order
    Unit Testing
    Integration Testing
    System testing
    Acceptance Testing
15. **Define Reverse Engineering. APR/MAY 2019**

    Reverse engineering is a process of de-constructing a system in order to extend the functionalities or in order to understand the working of the system

ANNA UNIVERSITY QUESTIONS

PART B

1. State the need for refactoring. How can a development model benefit by the use of refactoring? **(APR/MAY 2016) (NOV/DEC 2016)**
2. Why does software testing need extensive planning? Explain. **(APR/MAY 2016)**
3. Compare and contrast alpha and beta testing. **(APR/MAY 2016)**
4. Consider a program for determining the previous date. Its input is a triple of day, month and year with the values in the range $1 \leq \text{month} \leq 12$, $1 \leq \text{day} \leq 31$, $1990 \leq \text{year} \leq 2014$. The possible outputs would be previous date or invalid input date. Design the boundary value test cases. **(APR/MAY 2016)**

    Answer:

Note:
In Boundary Value Analysis, Min Min + Nominal Max – Max

Month [1-12]	Day [1-31]	Year [1990 – 2014]
Min :1 Min+ :2 Nominal : 6 Max - : 11 Max + : 12	1 2 15 30 31	1990 1991 2002 2013 2014

The boundary value test cases are

Test Case ID	Month (mm)	Day (dd)	Year (yyyy)	Expected Output
1	6	15	1990	14 June, 1990
2	6	15	1991	14 June, 1991
3	6	15	2002	14 June, 2002
4	6	15	2013	14 June, 2013
5	6	15	2014	14 June, 2014
6	6	1	2002	31 May, 2002
7	6	2	2002	1 June, 2002
8	6	30	2002	29 June, 2002
9	6	31	2002	Invalid Date as June has 30 Days
10	1	15	2002	14 January, 2002
11	2	15	2002	14 February, 2002
12	11	15	2002	14 November, 2002
13	12	15	2002	14 December, 2002

5. Describe the various Black box and White box testing techniques. Use Suitable examples for your explanation. **(APR/MAY 2015)**
6. Discuss about the various Integration and Debugging strategies followed in Software development. **(APR/MAY 2015)**
7. What is white box testing'? Explain. **(APR/MAY 2017)**
8. Explain how the various types of loops are tested. **(NOV/DEC 2017)**
9. **Differentiate black box and white box testing**. **(NOV/DEC 2017)**
10. What is black box testing? Explain the different types of black box testing strategies. Explain by considering suitable examples.

11. Explain unit testing and integration testing process with an example

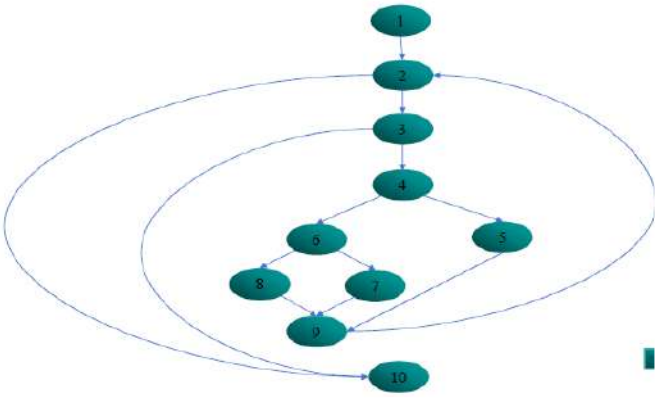
12. What is integration. testing? Discuss any one method in detail. **(APR/MAY 2017)**

13. Consider the following program segment. **(NOV/DEC 2017)**

```
/* num is the number the function searches in a presorted integer array arr */
Int bin_search(int num)
{
Int min, max; min = 0; max = 100;
While (min != max) {
If(arr [(min + max) / 2 > num]
Max = (min + max) / 2;
Else if (arr[( min + max)/2])
Min = (min + max) / 2;
Else return ( (min + max) / 2);
}
Return (-1);
}
```

(i)Draw the control flow graph for this program segment. (2)

Answer:



(ii) Define cyclomatic complexity. (2)

Answer:

Cyclomatic complexity is a software metric used to measure the complexity of a program.This metric measures independent paths through the program's source code. An independent path is defined as a path that has at least one edge which has not been traversed before in any other paths.

Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

(iii) Determine the cyclomatic complexity for this program. (Show the intermediate steps in your computation. Writing only the final result is not sufficient)

Answer:

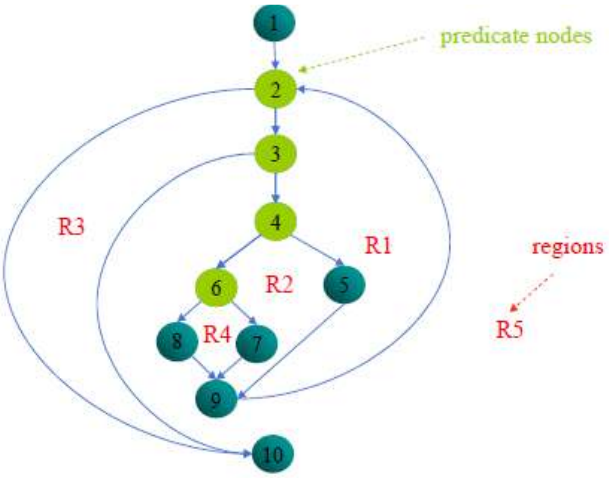
- Path 1: 1-2-10
- Path 2: 1-2-3-4-6-8-9-2-10
- Path 3: 1-2-3-4-6-8-9-2-3-10

} independent paths

- Path 4: 1-2-3-4-6-8-9-2-3-4-6-8-9-2-10 (not an independent path)

28

Three ways to compute cyclomatic complexity:
– The number of regions of the flow graph correspond to the cyclomatic complexity.
– Cyclomatic complexity, V(G), for a flow graph G is defined as V(G) = E - N + 2 (13-10+2=5) where E is the number of flow graph edges and N is the number of flow graph nodes.
– Cyclomatic complexity, V(G) = P + 1 where P is the number of predicate nodes contained in the flow graph G.

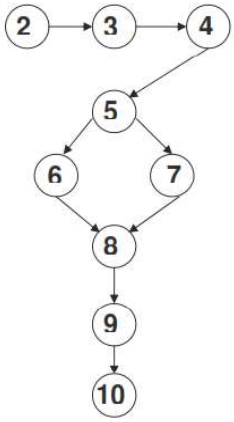


14. Consider the pseudocode for simple subtraction given below :

Program 'Simple Subtraction'
Input (x, y)
Output(x)
Output (y)
If x > y then
DO x y z
Else y x = z
Endlf
Output (z)
Output 'End Program"
Perform basic path testing and generate test cases. **(APR/MAY 2017) (NOV/DEC 2016)**

Answer:

Step 1:



Step 2: Compute Cyclomatic Complexity using formulas

V(G) = e – n + 2
= 9 – 9 + 2 = 2

Therefore we have to find 2 independence paths for basis path testing

Step 3:

Path 1-: 2-3-4-5-6-8-9-10

Path 2 -: 2-3-4-5-7-8-9-10

Step 4: Tes test cases for these paths are as given below

Independent path	**X**	**Y**	**Expected Result (z)**
Path 1 2-3-4-5-6-8-9-10	**10**	**5**	**5** **End program**
Path 2 2-3-4-5-7-8-9-10	**5**	**10**	**5** **End program**

15. Describe black box testing. Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m 1, cl) and (m2, c2) defining the two straight lines of the form y = mx + c. (APR/MAY 2017)

Answer:

The equivalence classes are the following:

•Parallel lines (m1=m2, c1≠c2)

•Intersecting lines (m1≠m2)

•Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7) , (10, 10) (10, 10) are obtained.

16. A program spacs state the following for an input field. The program shall accept an input value of 4-digit integer equal or greater than 2000 and less than or equal 8000. Determine the test cases using

2. **Equivalence class partitioning**
3. **Boundary value analysis**

Answer:

1. **Equivalence class partitioning:**

   Test cases for input box accepting numbers between 2000 and 8000 using Equivalence Partitioning:

   1) One input data class with all valid inputs. Pick a single value from range 2000 to 8000 as a valid test case. If you select other values between 2000 and 8000 the result is going to be same. So one test case for valid input data should be sufficient.

   2) Input data class with all values below the lower limit. I.e. any value below 2000, as an invalid input data test case.

   3) Input data with any value greater than 8000 to represent third invalid input class.

2. **Boundary value analysis**

   Test cases for input box accepting numbers between 1 and 1000 using Boundary value analysis:

   1) Test cases with test data exactly as the input boundaries of input domain i.e. values 2000 and 8000 in our case.

   2) Test data with values just below the extreme edges of input domains i.e. values 1999 and 7999.

   3) Test data with values just above the extreme edges of input domain i.e. values 2001 and 8000

17. Elaborate path testing and regression testing with an example**. NOV/DEC2019**

18. Explain how business process Reengineering helps to achieve a defined business outcome**. NOV/DEC2019**

19. Outline how the reverse engineering process helps to improve the legacy software. **NOV/DEC2019**

20. List the process in software reengineering process model and explain in detail. **APR/MAY 2019**

21. Write the procedure for the following: Given three sides of triangle, return the type of triangle that equilateral, isosceles, and scalene triangle. Draw the flow graph and calculate cyclomatic complexity to calculate the minimum number of paths. Enumerate the paths to be tested. **APR/MAY 2019**

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

5.1. ESTIMATION

- Software cost and effort estimation will never be an exact science.
- The variables such as human, technical, political, environmental can affect the ultimate cost of software and effort applied to develop it.
- To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project
2. Base estimates on similar projects that have already been completed.
3. Use relatively **simple decomposition techniques** to generate project cost and effort estimates.
4. Use one or more **empirical models** for software cost and effort estimation.

A model is based on experience (historical data) and takes the form

$d = f(v_i)$

where,

d is one of a number of estimated values (e.g., effort, cost, project duration)

v_i are selected independent parameters (e.g., estimated LOC or FP).

Decomposition techniques

1. FP based
2. LOC based

Empirical models

1. COCOMO-II model

5.1.1 Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

1. The degree to which you have properly estimated the size of the product to be built.
2. The ability to translate the size estimate into human effort, calendar time, and dollars.
3. The degree to which the project plan reflects the abilities of the software team.
4. The stability of product requirements and the environment that supports the software engineering effort.

Four different approaches to the sizing problem:

- ***"Fuzzy logic" sizing:***

  The planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

- ***Function point sizing:***

  The planner develops estimates of the information domain characteristics.

- ***Standard component sizing:***

  Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions.

The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.

- ***Change sizing:***

  This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type of modifications that must be accomplished.

5.1.2 Problem-Based Estimation:

LOC and FP data are used in two ways during software project estimation:

(1) As estimation variables to "size" each element of the software

(2) As baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

- **Baseline productivity metrics** (e.g., LOC/pm or FP/pm) are then applied to the appropriate estimation variable, and cost or effort for the function is derived.
- **Function estimates** are combined to produce an overall estimate for the entire project.

Using historical data or intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value.

The *expected value* for the estimation variable (size) *S* can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates.

$$S = \frac{S_{opt} + 4S_m + S_{pess}}{6}$$

5.2. Lines of Code (LOC)

- LOC metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program.
- Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.
- Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult.
- In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules and each module into sub modules and so on, until the sizes of the different leaf-level modules can be approximately predicted.
- To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

Advantages:

- LOC is the simplest among all metrics available to estimate project size.
- Many existing methods use LOC as a key input.
- A large body of literature and data based on LOC already exists.

Disadvantages:

1) LOC is dependent upon the programming language.
2) A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it.
3) This method is well designed but shorter program may get suffered.
4) It does not accommodate non procedural languages.

5) It is very difficult to accurately estimate LOC in the final. The LOC count can be accurately computed only after the code has been fully developed.

EXAMPLE: LOC APPROACH

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 (PC)	2,100
design analysis modules (DAM)	8,400
estimated lines of code	33,200

Average productivity for systems of this type = 620 LOC/pm and Burdened labor rate is Rs. 8000 per month. Find the total estimated project cost and effort.

Answer:

The cost per line of code = Cost / LOC = 8000/620 = Rs.13.

Based on the LOC estimate and the historical productivity data, the total estimated project cost is (33200*13) Rs. 431,000 and the estimated effort is 54 person-months

5.3 Function point (FP)

- This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.
- The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports.
- A software product supporting many features would certainly be of larger size than a product with less number of features.
- Each function when invoked reads some input data and transforms it to the corresponding output data.
- Besides using the number of input and output data values, function point metric computes the size of a software product using three other characteristics of the product. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics.
- The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

UFP = (Number of inputs)*4 + (Number of outputs)*5 +
(Number of inquiries)*4 + (Number of files)*10 +
(Number of interfaces)*10

Number of inputs:

✓ Each data item input by the user is counted. Data inputs should be distinguished from user inquiries.

Number of outputs:

- ✓ Each user output that provides application data to the user is counted. E.g. screens, reports, error messages.

Number of inquiries:

- ✓ Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

Number of files:

- ✓ Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

Number of interfaces:

- ✓ Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

- Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next.
- TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc.
- Each of these **14 factors** is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI).

  TCF = (0.65+0.01*DI)
- As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35.

  FP=UFP*TCF

Advantages:

- ➢ Function point metric can be used to easily estimate the size of a software product directly from the problem specification.

Disadvantages:

- ➢ Function point measure does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same.

<u>Feature point metric :</u>

- Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.
- Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.
- A three-point or expected value can then be computed. The *expected value* for the estimation variable (size) *S* can be computed as a weighted average of the optimistic (s_{opt}), most likely

$$S = \frac{S_{opt} + 4S_m + S_{pess}}{6} \qquad (5.1)$$

(s_m), and pessimistic (s_{pess}) estimates.

EXAMPLE: FP APPROACH

Information Domain Value	opt.	likely	pess.	est. count	weight	FP-count
number of inputs	20	24	30	24	4	97
number of outputs	12	15	22	16	5	78
number of inquiries	16	22	28	22	5	88
number of files	4	4	5	4	10	42
number of external interfaces	2	2	3	2	7	15
count-total						**321**

- The estimated number of FP is derived:

$$FP_{estimated} = \text{count total} \times [0.65 + 0.01 \times \Sigma(F_i)] = 375$$

-
- Organizational average productivity = 6.5 FP/pm.
- Burdened labor rate = $8000 per month, approximately $1230/FP.
- Based on the FP estimate and the historical productivity data, total estimated project cost is $461,000 and estimated effort is 58 person-months.

5.4. THE MAKE/BUY DECISION

- In many software application areas, it is very cheap to acquire rather than develop computer software.
- 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,
(3) Software may be custom built by an outside contractor to meet the purchaser's specifications.

- The steps involved in the acquisition of software are defined by the criticality of the software to be purchased and the end cost.
- In some cases (e.g., low-cost PC software), it is less expensive to purchase and experiment than to conduct a lengthy evaluation of potential software packages.

In the final analysis, the make/buy decision is made based on the following conditions:
(1) Will the delivery date of the software product be sooner than that for internally developed Software?
(2) Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally?
(3) Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support? These conditions apply for each of the acquisition options.

5.4.1. Creating a Decision Tree

The steps just described can be augmented using statistical techniques such as decision tree analysis.

- For example, Figure depicts a 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
(4) Contract the software development to an outside vendor

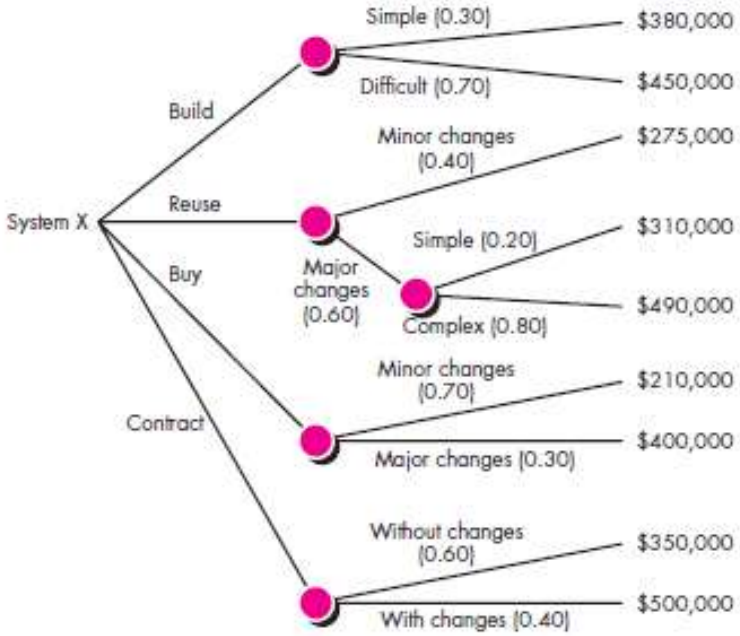


Fig. A decision tree to support the make/buy decision

- If the system is to be built from scratch, there is a 70 percent probability that the job will be difficult. Using the estimation techniques discussed earlier in this chapter, the 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,

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

- Following other paths of the decision tree, the projected costs for reuse, purchase, and contract, under a variety of circumstances, are also shown. The expected costs for these paths are

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

- Based on the probability and projected costs that have been noted, 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.

5.4.2. Outsourcing

- Sooner or later, every company that develops computer software asks a fundamental question: "Is there a way that we can get the software and systems we need at a lower price?" The answer to this question is not a simple one, and the emotional discussions that occur in response to the question always lead to a single word: *outsourcing.*
- In concept, outsourcing is extremely simple. Software engineering activities are contracted to a third party who does the work at lower cost and, hopefully, higher quality. Software work conducted within a company is reduced to a contract management activity.
- The decision to outsource can be either strategic or tactical.
- At the strategic level, business managers consider whether a significant portion of all software work can be contracted to others.
- At the tactical level, a project manager determines whether part or all of a project can be best accomplished by subcontracting the software work.
- Regardless of the breadth of focus, the outsourcing decision is often a financial one.

Pros:

- Cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them.

Cons:

- A company loses some control over the software that it needs. Since software is a technology that differentiates its systems, services, and products, a company runs the risk of putting the fate of its competitiveness into the hands of a third party.

5.5 . COCOMO MODEL

Any software development project can be classified into one of the following three categories based on the development complexity:

1) Organic
2) Semidetached
3) Embedded

1) Organic:

- A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

2) Semidetached:

- A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

3) Embedded:

- A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

COCOMO

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages:

(1) Basic COCOMO
(2) Intermediate COCOMO
(3) and Complete COCOMO

(1) Basic COCOMO Model :

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Effort = a_1 x (KLOC)^{a2} PM

Tdev = b_1 x (Effort)^{b2} Months

Where

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
- a_1, a_2, b_1, b_2 are constants for each category of software products,
- Tdev is the estimated time to develop the software, expressed in months,
- Effort is the total effort required to develop the software product, expressed in person months (PMs).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot. It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve.

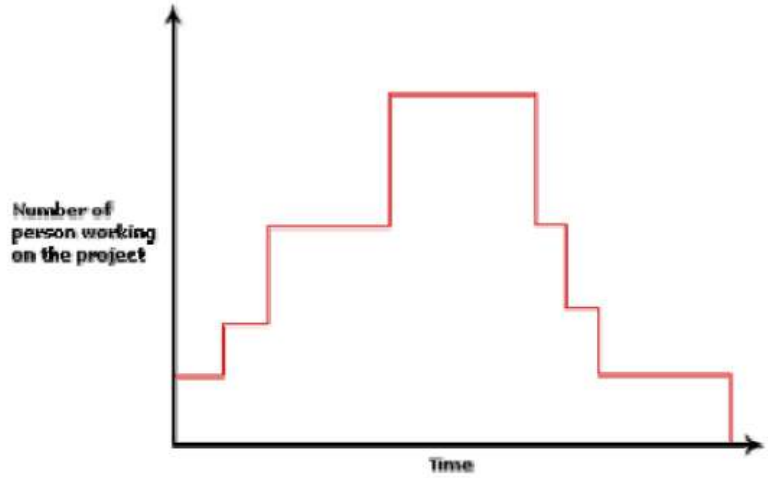


Fig. Person-month curve

Every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines, it is considered to be nLOC. The values of a_1, a_2, b_1, b_2 for different categories of products (i.e. organic, semidetached, and embedded) are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects.

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : **Effort = 2.4(*KLOC*)$^{1.05}$ PM**

Semi-detached : **Effort = 3.0(*KLOC*)$^{1.12}$ PM**

Embedded : **Effort = 3.6(*KLOC*)$^{1.20}$ PM**

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : **Tdev = 2.5(*Effort*)$^{0.38}$ Months**

Semi-detached : **Tdev = 2.5(*Effort*)$^{0.35}$ Months**

Embedded : **Tdev = 2.5(*Effort*)$^{0.32}$ Months**

- Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig. 11.4 shows a plot of estimated effort versus product size. From fig. 11.4, we can observe that the effort is somewhat super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

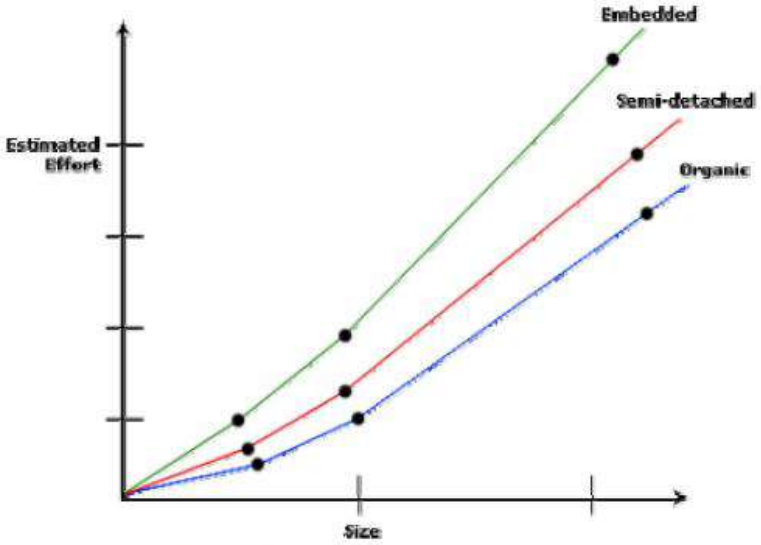


Fig. 11.4: Effort versus product size

- The development time versus the product size in KLOC is plotted in fig. 11.5. From fig. 11.5, it can be observed that the development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately.
- This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers.
- This reduces the time to complete the project. Further, from fig. 11.5, it can be observed that the development time is roughly the same for all the three categories of products.
- It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate.
- The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically.
- But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

Example:

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = **2.4 x (32)$^{1.05}$ = 91 PM**

Nominal development time = **2.5 x (91)$^{0.38}$ = 14 months**

Cost required to develop the product = **14 x 15,000**

= **Rs. 210,000/-**

(2) Intermediate COCOMO model :

- The intermediate COCOMO model recognizes refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development.
- If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. The project manager to rate these 15 different parameters for a particular project on a scale of one to three.
- Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO.

In general, the cost drivers can be classified as being attributes of the following items:

- Product attributes
  - Required software reliability
  - Size of application database
  - Complexity of the product
- Hardware attributes
  - Run-time performance constraints
  - Memory constraints
  - Volatility of the virtual machine environment
  - Required turnabout time
- Personnel attributes
  - Analyst capability
  - Software engineering capability
  - Applications experience
  - Virtual machine experience
  - Programming language experience
- Project attributes
  - Use of software tools
  - Application of software engineering methods
  - Required development schedule

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

The Intermediate COCOMO formula now takes the form:

$Effort=a_1(KLOC)^{a2}.EAF$

$Tdev=b1(Effort)^{b2}$ Months

(3) Complete COCOMO model :

- A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics.
- The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.
- The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:
  - Database part
  - Graphical User Interface (GUI) part
  - Communication part

5.6 COCOMO II Model

- 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.
- Like its predecessor, **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:
  - Object points, Function points and Lines of code(LOC).
- The COCOMO II application composition model uses object points.
- 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
(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).
- In essence, 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.
- Once complexity is determined, the number of screens, reports, and components are weighted according to the table illustrated in Figure .

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

FIG. Complexity weighting for object types.

- The object point count is then determined by multiplying the original number of object instances by the weighting factor in the figure and summing to obtain a total object point count.
- 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.

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

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

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

In more advanced COCOMO II models, a variety of scale factors, cost drivers,and adjustment procedures are required.

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

FIG. Productivity rate for object points.

Example:

Describe in detail COCOMO model for software cost estimation. Use it to estimate the effort required to build software for a simple ATM that produces 12 screens, 10 reports and has 80 software components. Assume average complexity and average developer maturity. Use application composition model with object points. (NOV/DEC 2016)

Answer:

COCOMO Model Explanation and answer for problem need to write

Formula Note:

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

Object Point = (Screen * Weighting factor) + (Report * Weighting factor) + (Component * Weighting factor)

NOP = (Object Points) * $\frac{(100 - \%\ reuse)}{100}$

PROD = $\frac{NOP}{Person/Month}$

Estimated Effort = $\frac{NOP}{PROD}$

Productivity Rate for Object point:

Developer Experience Environment Maturity / Capability	Very Low Very Low	Low Low	Nominal Nominal	High High	Very High Very High
PROD	4	7	13	25	50

Solution for question:

Object Type	Count	Complexity Weight		
		Simple	Medium	Difficult
Screen	12		2	
Report	10		5	
3 GL Component	80			10

Object Point = 12 *2 + 10*5 + 80*10 = 874

Assume 80% of reuse

$$\textbf{NOP} = (\text{Object Points}) * \frac{(100 - \%\ \text{reuse})}{100}$$

= 874 * {(100 -80)/100}

= 874*0.2

NOP = 174.8

Nominal Developer Experience

So, <u>PROD = 13</u>

$$\textbf{Estimated Effort} = \frac{\text{NOP}}{\text{PROD}}$$

= 174.8/13

Estimated Effort = 13.45

5.7. SCHEDULING AND TRACKING

- *Software project scheduling* is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. During early stages of project planning, a macroscopic schedule is developed.
- This type of schedule identifies all major process framework activities and the product functions to which they are applied. Here, <u>specific software actions and tasks are identified and scheduled.</u>
- Scheduling for software engineering projects can be viewed from two rather different perspectives.

  1. In the first, an end date for release of a computer-based system has already (and irrevocably) been established.

  2. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization.

Basic Principles

1) ***Compartmentalization***. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.
2) ***Interdependency.*** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

3) ***Time allocation***. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.
4) ***Effort validation***. Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time.
5) ***Defined responsibilities***. Every task that is scheduled should be assigned to a specific team member.
6) ***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. Work products are often combined in deliverables.
7) ***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 and has been approved.

Each of these principles is applied as the project schedule evolves.

5.7.1 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.

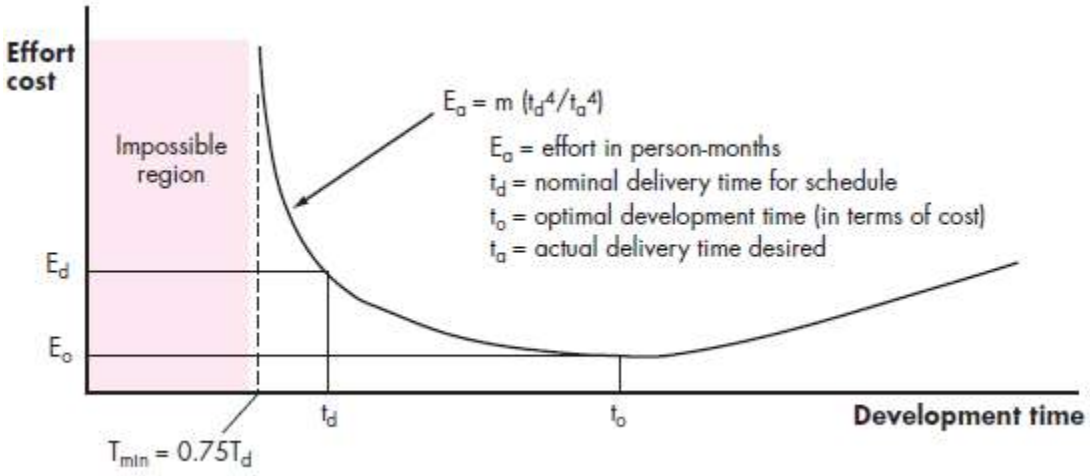


Fig. The relationship between effort and delivery time

- There is a common myth that is still believed by many managers who are responsible for software development projects: "If we fall behind schedule, we can always add more programmers and catch up later in the project." Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work. While teaching, no work is done, and the project falls further behind.
- **The *Putnam-Norden-Rayleigh (PNR) Curve*** provides an indication of the relationship between effort applied and delivery time for a software project. A version of the curve, representing project effort as a function of delivery time, is shown in Figure 27.1. The

curve indicates a minimum value *to* that indicates the least cost for delivery (i.e., the delivery time that will result in the least effort expended). As we move left of *to* (i.e., as we try to accelerate delivery), the curve rises nonlinearly.

- As an example, we assume that a project team has estimated a level of effort *Ed* will be required to achieve a nominal delivery time *td* that is optimal in terms of schedule and available resources. Although it is possible to accelerate delivery, the curve rises very sharply to the left of *td*. In fact, the PNR curve indicates that the project delivery time cannot be compressed much beyond 0.75*td*. If we attempt further compression, the project moves into "the impossible region" and risk of failure becomes very high. The PNR curve also indicates that the lowest cost delivery option,
- 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*:

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

where *E* is the effort expended (in person-years) over the entire life cycle for software development and maintenance and *t* is the development time in years.

The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year).

This leads to some interesting results. Consider a complex, real-time software project estimated at 33,000 LOC, 12 person-years of effort. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years, the highly nonlinear nature of the model described in Equation (27.1) yields:

E _ ~ 3.8 person-years

- This implies that, by extending the end date by six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear: benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

Effort Distribution

- Each of the software project estimation techniques leads to estimates of work units (e.g., person-months) required to complete software development. 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. You can correctly infer that coding (20 percent of effort) is deemphasized.
- This effort distribution should be used as a guideline only.

5.7.2. A Task Set for the Software Project

- A task set is a collection of software engineering work tasks, milestones, work products, and quality assurance filters that must be accomplished to complete a particular project. The task set must provide enough discipline to achieve high software quality. But, at the same time, it must not burden the project team with unnecessary work.
- In order to develop a project schedule, a task set must be distributed on the project time line. The task set will vary depending upon the project type and the degree of rigor with which the software team decides to do its work.
- Software organizations encounter the following projects:

1. ***Concept development projects*** that are initiated to explore some new business concept or application of some new technology.

2. ***New application development*** **projects** that are undertaken as a consequence of a specific customer request.

3. ***Application enhancement*** **projects** that occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.

4. ***Application maintenance projects*** that correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.

5. ***Reengineering projects*** that are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

- Even within a single project type, many factors influence the task set to be chosen.
- These include: size of the project, number of potential users, mission criticality, application longevity, stability of requirements, ease of customer/developer communication, maturity of applicable technology, performance constraints, embedded and non embedded characteristics, project staff, and reengineering factors.
- When taken in combination, these factors provide an indication of the *degree of rigor* with which the software process should be applied.

A Task Set Example

- Concept development projects are initiated when the potential for some new technology must be explored. There is no certainty that the technology will be applicable, but a customer (e.g., marketing) believes that potential benefit exists. Concept development projects are approached by applying the following actions:

1.1 Concept scoping determines the overall scope of the project.

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

1.3 Technology risk assessment evaluates the risk associated with the technology to be implemented as part of the project scope.

1.4 Proof of concept demonstrates the viability of a new technology in the software context.

1.5 Concept implementation implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.

1.6 Customer reaction to the concept solicits feedback on a new technology concept and targets specific customer applications.

5.7.3. Defining a Task Network:

- A *task network,* also called an *activity network,* is a graphic representation of the task flow for a project.
- It is sometimes used as the mechanism through which task sequence and dependencies are input to an automated project scheduling tool.
- In its simplest form (used when creating a macroscopic schedule), the task network depicts major software engineering actions. Figure shows a schematic task network for a concept development project.
- The concurrent nature of software engineering actions leads to a number of important scheduling requirements. Because parallel tasks occur asynchronously, you should determine intertask dependencies to ensure continuous progress toward completion.
- In addition, you should be aware of those tasks that lie on the *critical path.* That is, tasks that must be completed on schedule if the project as a whole is to be completed on schedule.

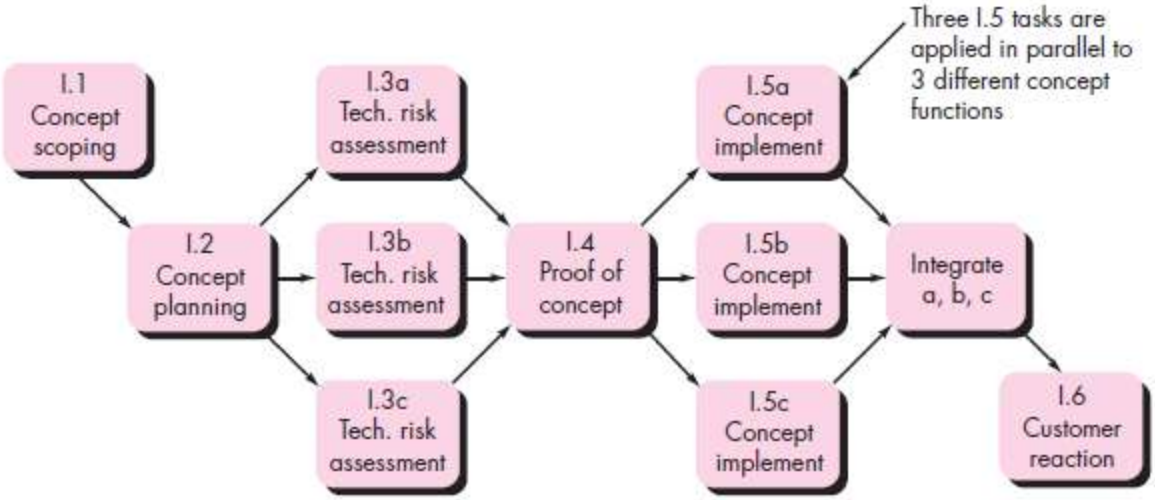


Fig. A task network for concept development

5.7.4. 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.
- The project scheduling methods that can be applied to software development are
  1. *Program evaluation and review technique* (PERT)
  2. *critical path method* (CPM)
- Both techniques are driven by information already developed in earlier project planning activities:
  Estimates of effort
  A decomposition of the product function
  The selection of the appropriate process model and task set
  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.

5.7.4.1 Time-Line Charts

- When creating a software project schedule, you begin with a set of tasks (the work breakdown structure).
- If automated tools are used, the work breakdown is input as a task network or task outline.
- Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.
- As a result of this input, a *time-line chart*, also called a *Gantt chart*, is generated.
- A time-line chart can be developed for the entire project.
- Alternatively, separate charts can be developed for each project function or for each individual working on the project.
- Figure illustrates the format of a time-line chart.
- It depicts a part of a software project schedule that emphasizes the concept scoping task for a word-processing (WP) software product.
- All project tasks (for concept scoping) are listed in the lefthand column.
- The horizontal bars indicate the duration of each task.
- When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.

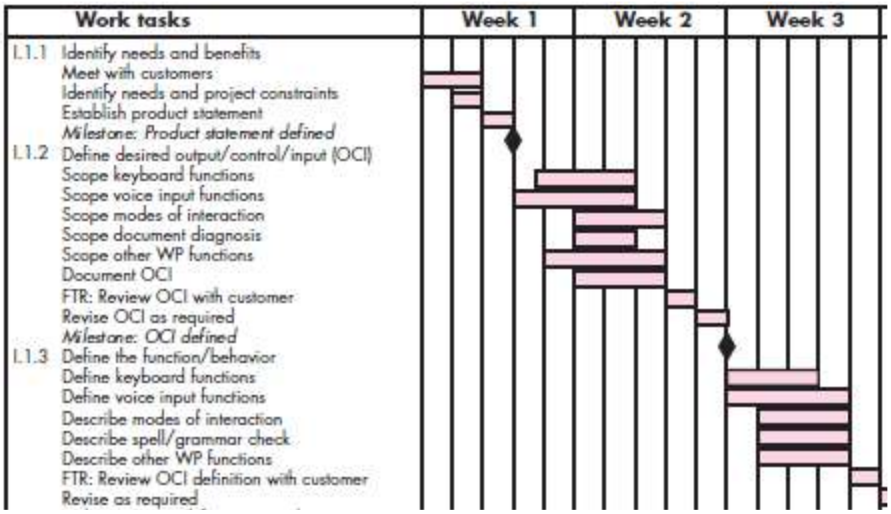


Fig. An example time-line chart

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
1.1.1 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			
1.1.2 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				
1.1.3 Define the function/behavior							

Fig. An example project table Tracking the Schedule

5.7.5. Tracking the schedule:

- 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:
  - Conducting **periodic project status meetings** in which each team member reports progress and problems
  - **Evaluating the results of all reviews** conducted throughout the software engineering process
  - Determining whether **formal project milestones** (the diamonds shown in Figure) have been accomplished by the scheduled date
  - Comparing the actual start date to the planned start date for each project task listed in the resource table
  - **Meeting informally with practitioners** to obtain their subjective assessment of progress to date and problems on the horizon
  - Using earned **value analysis** 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.

Tracking Progress for an OO Project

- Although an iterative model is the best framework for an OO project, task parallelism makes project tracking difficult.
- In general, the following major milestones can be considered "completed" when the criteria noted have been met.

Technical milestone: OO analysis completed

• All classes and the class hierarchy have been defined and reviewed.
• Class attributes and operations associated with a class have been defined and reviewed.
• Class relationships have been established and reviewed.
• A behavioral model has been created and reviewed.
• Reusable classes have been noted.

Technical milestone: OO design completed

• The set of subsystems has been defined and reviewed.
• Classes are allocated to subsystems and reviewed.
• Task allocation has been established and reviewed.
• Responsibilities and collaborations have been identified.
• Attributes and operations have been designed and reviewed.
• The communication model has been created and reviewed.

Technical milestone: OO programming completed

• Each new class has been implemented in code from the design model.
• Extracted classes (from a reuse library) have been implemented.
• Prototype or increment has been built.**Technical milestone: OO testing**
• The correctness and completeness of OO analysis and design models has been reviewed.
• A class-responsibility-collaboration network has been developed and reviewed.
• Test cases are designed, and class-level tests have been conducted for each class.
• Test cases are designed, and cluster testing is completed and the classes are integrated.
• System-level tests have been completed.

5.8. EARNED VALUE ANALYSIS

- A technique for performing quantitative analysis of progress is called earned value analysis (EVA).
- 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.
- Stated even more simply, earned value is a measure of progress. It enables you to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling.
- Fleming and Koppleman argue that earned value analysis "provides accurate and reliable readings of performance from as early as 15 percent into the project."

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

1) The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned. Hence, BCWSi is the effort planned for work task i. To determine progress at a given point along the project

schedule, the value of BCWS is the sum of the BCWSi values for all work tasks that should have been completed by that point in time on the project schedule.

2) The BCWS values for all work tasks are summed to derive the budget at completion (BAC). Hence,

$$BAC = \Sigma\ (BCWS_k) \text{ for all tasks } k$$

3) Next, the value for budgeted cost of work performed (BCWP) 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.

- The distinction between the BCWS and the BCWP is that the former represents the budget of the activities that were planned to be completed and the latter represents the budget of the activities that actually were completed." Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

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

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

- SPI is an indication of the efficiency with which the project is utilizing scheduled resources. An SPI value close to 1.0 indicates efficient execution of the project schedule. SV is simply an absolute indication of variance from the planned schedule.

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

It Provides an indication of the percentage of work that should have been completed by time t.

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

It provides a quantitative indication of the percent of completeness of the project at a given point in time t.

It is also possible to compute the actual cost of work performed (ACWP). The value for ACWP is the sum of the effort actually expended on work tasks that have been completed by a point in time on the project schedule. It is then possible to compute

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

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

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.

Earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.

5.9. PROJECT PLAN

Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail. **A project plan** is created that records the work to be done, who will do it, the development schedule, and the work products.

Project and organization, plans normally include the following sections:

1. **Introduction** This briefly describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the management of the project.

2. **Project organization** This describes the way in which the development team is organized, the people involved, and their roles in the team.
3. **Risk analysis** This describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies that are proposed.
4. **Hardware and software resource requirements** This specifies the hardware and support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.
5. **Work breakdown** This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity. Milestones are key stages in the project where progress can be assessed; deliverables are work products that are delivered to the customer.
6. **Project schedule** This shows the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities. The ways in which the schedule may be presented are discussed in the next section of the chapter.
7. **Monitoring and reporting mechanisms** This defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

5.10. THE PLANNING PROCESS

Project planning is an iterative process that starts when you create an initial project plan during the project startup phase. UML activity diagram shows a typical workflow for a project planning process. Plan changes are inevitable.

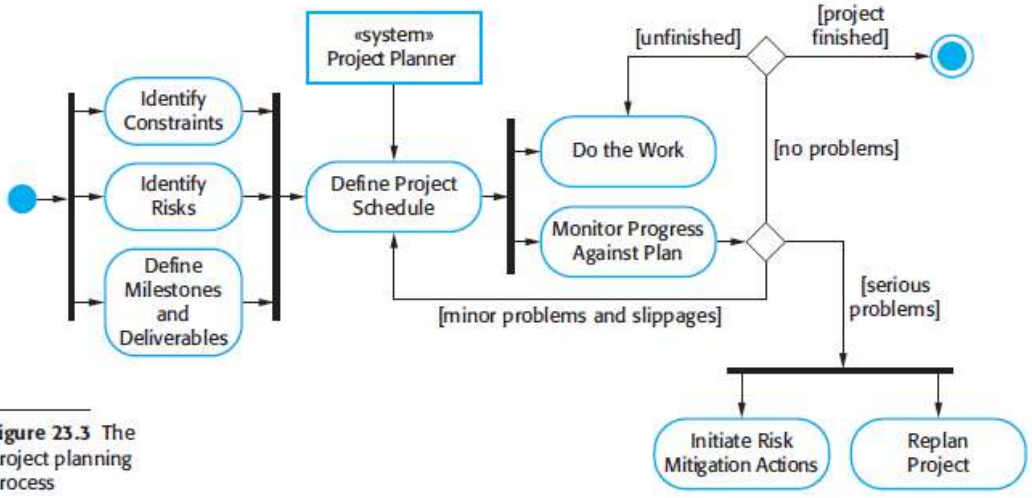


Figure 23.3 The project planning process

- At the beginning of a planning process, you should assess the constraints affecting the project. These constraints are the required delivery date, staff available, overall budget, available tools, and so on.
- In conjunction with this, you should also identify the project milestones and deliverables. Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing.
- Deliverables are work products that are delivered to the customer (e.g., a requirements document for the system).
- The process then enters a loop. You draw up an estimated schedule for the project and the activities defined in the schedule are initiated or given permission to continue.
- The outcome of a review may be a decision to cancel a project. This may be a result of technical or managerial failings but, more often, is a consequence of external changes that affect the project. The development time for a large software project is often several years.

5. 8 . RISK MANAGEMENT

Risks – Definition

The risk denotes the uncertainty that may occur in the choices due to past actions and risk is something which causes heavy losses.

Definition of risk Management:

Risk management refers to the process of making decisions based on an evalution of factors that threats to the business.

Various activities that are carried out for risk management are-

1. Risk Identification 2. Risk Projection 3. Risk Refinement 4. Risk Mitigation, Monitoring and Management

Reactive Risk Management

- project team reacts to risks when they occur
- mitigation—plan for additional resources in anticipation of fire fighting
- fix on failure—resource are found and applied when the risk strikes
- crisis management—failure does not respond to applied resources and project is in jeopardy

Proactive Risk Management

- formal risk analysis is performed
- organization corrects the root causes of risk
  - ✓ TQM concepts and statistical SQA
  - ✓ examining risk sources that lie beyond the bounds of the software
  - ✓ developing the skill to manage change

Software Risks

There are two characteristics of risks

1. The risk may or may not happen. It shows uncertainty of the risks.
2. When risks occur, unwanted consequences or losses will occur.

Types of risks:

1. Project risks:

   Project risks arise in the software development process then they basically affect budget, schedule, staffing, resources and requirements.

When project risks become severe then the total cost of project gets increased.

2. Technical risks:

   These risks affect quality and timeliness of the project. If technical risks become reality then potential design implementation, interface, verification and maintenance problems gets created. Technical risks occur when problem becomes harder to solve.

3. Business risks

   When feasibility of software product is in suspect then business risks occur. Business risks can be further categorized as

   i. Marker risk
   ii. Strategic risk
   iii. Sales risk
   iv. Management risk
   v. Budget risk

4. Known risks

   These are identified after evaluating the project plan. These risks can be identified from other sources such as environment in which the product get developed, unrealistic deadlines, poor requirement specification and software scope. There are two types of known risks – **Predictable and unpredictable risks.**

   Predictable risks Those risks that can be identified in advance based on past experience

   Unpredictable risks are those that cannot be guessed earlier.

Risk Management Paradigm

5.9. Risk Identification

Risk identification can be defined as the efforts taken to specify threats to the project plan**.** Risk identification can be done bby identifying the known and predictable risks.

- *Product size*—risks associated with the overall size of the software to be built or modified.
- *Business impact*—risks associated with constraints imposed by management or the marketplace.
- *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.

- *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.
- *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.
- *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

Assessing Project Risk-I

- Have top software and customer managers formally committed to support the project?
- Are end-users enthusiastically committed to the project and the system/product to be built?
- Are requirements fully understood by the software engineering team and their customers?
- Have customers been involved fully in the definition of requirements?
- Do end-users have realistic expectations?

Assessing Project Risk-II

- Is project scope stable?
- Does the software engineering team have the right mix of skills?
- Are project requirements stable?
- Does the project team have experience with the technology to be implemented?
- Is the number of people on the project team adequate to do the job?
- Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk Components

- *performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- *cost risk*—the degree of uncertainty that the project budget will be maintained.
- *support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- *schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

5.10.Risk Projection

- *Risk projection,* also called *risk estimation,* attempts to rate each risk in two ways
  - ✓ the likelihood or probability that the risk is real
  - ✓ the consequences of the problems associated with the risk, should it occur.
- The are four risk projection steps:
  - ✓ establish a scale that reflects the perceived likelihood of a risk
  - ✓ delineate the consequences of the risk
  - ✓ estimate the impact of the risk on the project and the product,
  - ✓ note the overall accuracy of the risk projection so that there will be no misunderstandings.

Building a Risk Table

- A risk table provides us with a simple technique for risk projection

- A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time.
- High-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.
- All risks that lie above the cutoff line should be managed.
- Risk probability can be determined by making individual estimates and then developing a single consensus value.
- Risk drivers can be assessed on a qualitative probability scale that has the following values: impossible, improbable, probable, and frequent.

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

Figure 5.10 Sample risk table prior to sorting

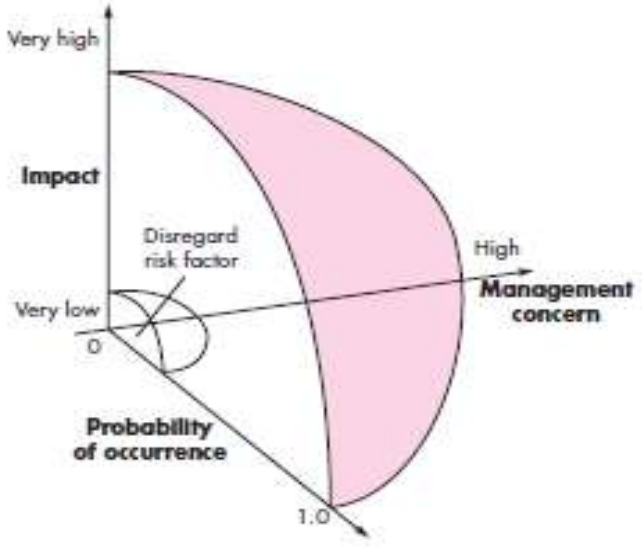


Figure 5.11 Risk and management concern

Assessing Risk Impact:

Three factors affect the consequences of a risk

i) Nature ii) Scope iii) Timing

- The nature of the risk indicates the problems that are likely if it occurs.
- The scope of a risk combines the severity with its overall distribution
- Finally, the timing of a risk considers when and for 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 shown.
  (3) Complete the risk table and analyze the results.
- The overall ***risk exposure***, RE, is determined using the following relationship [Hal98]:

  RE = *P* x *C*

where

P is the probability of occurrence for a risk, and *C* is the cost to the project should the risk occur.

5.11. RISK MITIGATION, MONITORING, AND MANAGEMENT (RMMM)

- An effective strategy must consider three issues:
  i) Risk avoidance
  ii) Risk monitoring
  iii) Risk management and contingency planning.
- If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation.*
- To mitigate this risk, you would develop a strategy 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, and 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.
- 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.
- In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps
- *Risk management and contingency planning* assumes that mitigation efforts have failed and that the risk has become a reality.
- For a large project, you should adapt the Pareto 80–20 rule to software risk.
- Experience indicates that 80 percent of the overall project can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help you to determine which of the risks reside in that 20.
- *Software safety and hazard analysis* are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.

THE RMMM PLAN

- A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate *risk mitigation, monitoring, and management plan* RMMM).
- 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.
- Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a *risk information sheet* (RIS) .
- The RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.
- 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 with three primary objectives:

(1) To assess whether predicted risks do, in fact, occur;

(2) To ensure that risk aversion steps defined for the risk are being properly applied; and

(3) To collect information that can be used for future risk analysis.

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

9.12 CASE TOOLS:

What is CASE?

The Computer Adied Software Engineering (CAES) tools automate the project management activities, manage all the work products.

Importance of CASE Tools

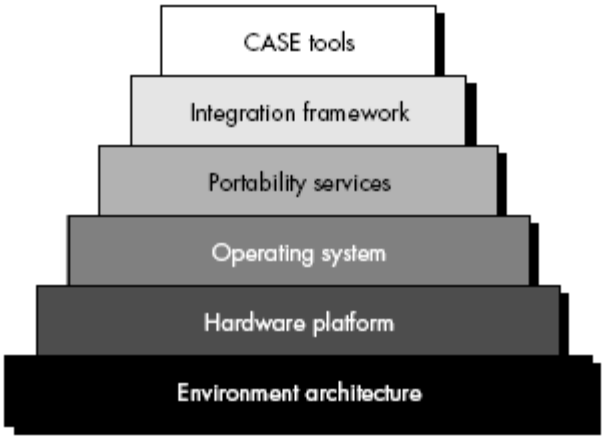
CASE provides the software engineer with the ability to automate manual activities and to improve engineering insight. Like computer-aided engineering and design tools that are used by

engineers in other disciplines, CASE tools help to ensure that quality is designed in before the product is built.

9.12.1. BUILDING BLOCKS FOR CASE

- Computer aided software engineering can be as simple as a single tool that supports a specific software engineering activity or as complex as a complete "environment" that encompasses tools, a database, people, hardware, a network, operating systems, standards, and myriad other components. This communication creates an **integrated environment**.

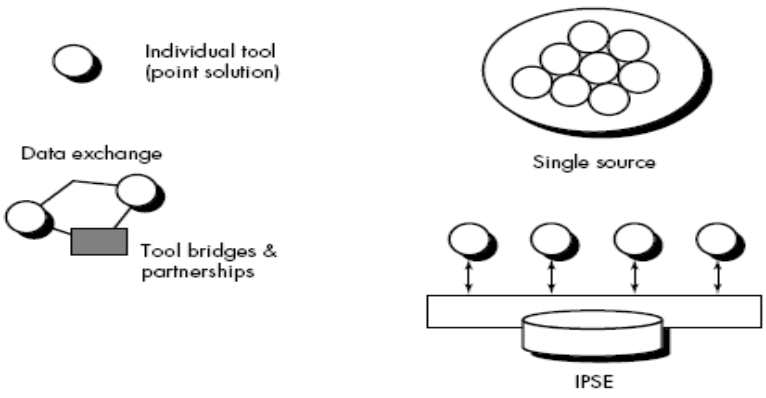
FIGURE : CASE building blocks



The bottom most layer consists of environment architecture and hardware platform. The environment architecture consists of collection of system software and human work pattern that is applied during the software engineering process.

- A set of Probability services connects the CASE tools with the integration framework.
- The integration framework is a collection of specialized programs which allows the CASE tools to communicate with the database and to create same look and feel for the end-user. Using probability services CASE tools can communicate with the cross platform elements.
- At the top of this building block a collection of CASE tools exist. CASE tools basically assist the software engineer in developing a complex component.
- CASE tools can exist in variety of manner. A single CASE tool can be used, or a collection of CASE tools may exists which acts as some package. The CASE tools may serve as a bridge between other tools.

FIGURE :Integration options



At the high end of the integration spectrum is the integrated project support environment (IPSE). Standards for each of the building blocks have been created. CASE tool vendors use IPSE standards to build tools that will be compatible with the IPSE and therefore compatible with one another.

5.12.1 A TAXONOMY OF CASE TOOLS

To create an effective CASE environment, various categories of tools can be developed.
CASE tools can be classified by

1. By function or use
2. By user type
3. By stage in software engineering process

The taxonomy of CASE tools is given below.

1. **Business process engineering tools.**

The primary objective for tools in this category is to represent business data objects, their relationships, and how these data objects flow between different business areas within a company.

2. **Process modeling and management tools**.

If an organization works to improve a business (or software) process, it must first understand it. Process modeling tools (also called process technology tools) are used to represent the key elements of a process so that it can be better understood.

3. **Project planning tools.**

Tools in this category focus on two primary areas: software project effort and cost estimation and project scheduling.

4. **Risk analysis tools.**

Identifying potential risks and developing a plan to mitigate, monitor, and manage them is of paramount importance in large projects. Risk analysis tools enable a project manager to build a risk table by providing detailed guidance in the identification and analysis of risks.

5. **Project management tools.**

The project schedule and project plan must be tracked and monitored on a continuing basis.. Tools in the category are often extensions to project planning tools.

6. **Requirements tracing tools**.

The objective of requirements tracing tools is to provide a systematic approach to the isolation of requirements, beginning with the customer request for proposal or specification.

7. **Metrics and management tools.**

Software metrics improve a manager's ability to control and coordinate the software engineering process and a practitioner's ability to improve the quality of the software that is produced.

8. **Documentation tools.**

Most software development organizations spend a substantial amount of time developing documents, and in many cases the documentation process itself is quite inefficient. For this reason, documentation tools provide an important opportunity to improve productivity.

9. **System software tools.**

CASE is a workstation technology. Therefore, the CASE environment must accommodate high-quality network system software, object management services, distributed component support, electronic mail, bulletin boards, and other communication capabilities.

10. Quality assurance tools.

The majority of CASE tools that claim to focus on quality assurance are actually metrics tools that audit source code to determine compliance with language standards. Other tools extract technical metrics in an effort to project the quality of the software that is being built.

11. Database management tools.

Database management software serves as a foundation for the establishment of a CASE database (repository) that we have called the project database.

12. Software configuration management tools.

Software configuration management lies at the kernel of every CASE environment. Tools can assist in all five major SCM tasks—identification, version control, change control, auditing, and status accounting.

13. Analysis and design tools.

The models contain a representation of data, function, and behavior (at the analysis level) and characterizations of data, architectural, component-level, and interface design.

14. PRO/SIM tools. PRO/SIM (prototyping and simulation) tools

It provide the software engineer with the ability to predict the behavior of a real-time system prior to the time that it is built.

15. Interface design and development tools.

Interface design and development tools are actually a tool kit of software components (classes) such as menus, buttons, window structures, icons, scrolling mechanisms, device drivers, and so forth.

16. Prototyping tools.

A variety of different prototyping tools can be used. Screen painters enable a software engineer to define screen layout rapidly for interactive applications.

17. Programming tools.

The programming tools category encompasses the compilers, editors, and debuggers that are available to support most conventional programming languages.

18. Web development tools.

The activities associated with Web engineering are supported by a variety of tools for WebApp development. These include tools that assist in the generation of text, graphics, forms, scripts, applets, and other elements of a Web page.

19. Integration and testing tools.

In their directory of software testing tools, Software Quality Engineering defines the following testing tools categories:

- Data acquisition.
- Static measurement
- Dynamic measurement
- Simulation
- Test management
- Cross-functional tools

20. Static analysis tools.

Static testing tools assist the software engineer in deriving test cases. Three different types of static testing tools are used in the industry:

i. Code-based testing tools accept source code (or PDL) as input and perform a number of analyses that result in the generation of test cases.

ii. Specialized testing languages (e.g., ATLAS) enable a software engineer to write detailed test specifications that describe each test case and the logistics for its execution.

iii. Requirements-based testing tools isolate specific user requirements and suggest test cases (or classes of tests) that will exercise the requirements.

21. Dynamic analysis tools.

Dynamic testing tools interact with an executing program, checking path coverage, testing assertions about the value of specific variables, and otherwise instrumenting the execution flow of the program.

22. Test management tools.

Tools in this category manage and coordinate regression testing, perform comparisons that ascertain differences between actual and expected output, and conduct batch testing,test management tools also serve as generic test drivers.

23.Client/server testing tools.

The c/s environment demands specialized testing tools that exercise the graphical user and the network communications requirements for client and server.

23. Reengineering tools.

Tools for legacy software address a set of maintenance activities that currently absorb a significant percentage of all software-related effort. The reengineering tools category can be subdivided into the following functions:

- Reverse engineering
- Code restructuring
- On-line system reengineering

5.11.2 Workbenches:

CASE workbench is a set of tools which supports a particular phase in the software process.
These tools work together to provide the complete support to the software development activity.
In the workbench common services are provided which are used by all the tools. Some kind of data integration is also supported.

Advantages:

1. it is available at low cost
2. there is productivity improvement dur to support of workbenches
3. it results in standardized documentation for software system

Drawbacks:

1. These systems are closed environments with tight integration with tools.
2. The import and export facilities for various types of data are limited.
3. It is difficult to adapt method for specific organizational need.

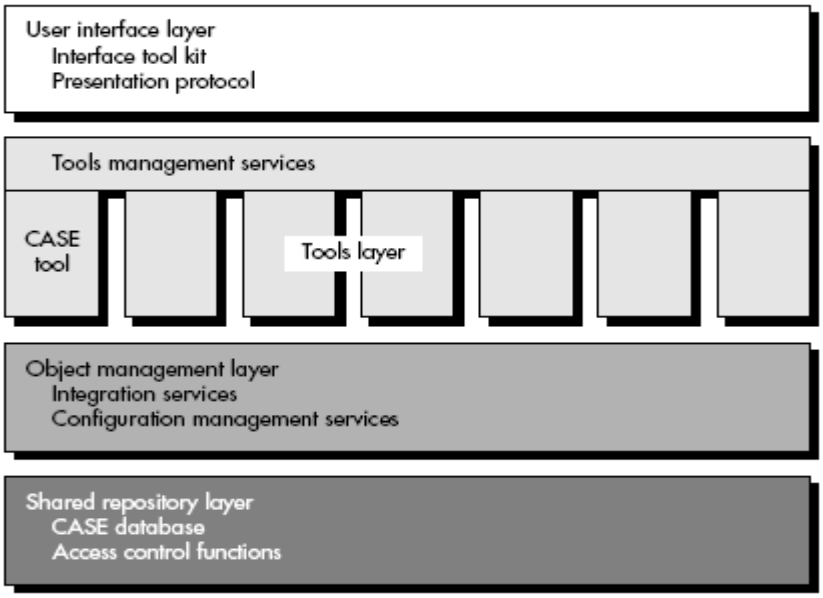
5.11.3. Integrated CASE Environment:

- A software engineering team uses CASE tools, corresponding methods, and a process framework to create a pool of software engineering information.
- The integration framework facilitates transfer of information into and out of the pool.
- To accomplish this, the following architectural components must exist: a database must be created (to store the information); an object management system must be built (to manage changes to the information); a tools control mechanism must be constructed (to coordinate the use of CASE tools); a user interface must provide a consistent pathway between actions

made by the user and the tools contained in the environment. Most models of the integration framework represent these components as layers.

- The user interface layer (in Figure) incorporates a standardized interface tool kit with a common presentation protocol.
- The interface tool kit contains software for human/computer interface management and a library of display objects. Both provide a consistent mechanism for communication between the interface and individual CASE tools.
- The presentation protocol is the set of guidelines that gives all CASE tools the same look and feel. Screen layout conventions, menu names and organization, icons, object names, the use of the keyboard and mouse, and the mechanism for tools access are all defined as part of the presentation protocol.

FIGURE :Architectural model for the integration framework



- The object management layer (OML) performs the configuration management functions.
- The shared repository layer is the CASE database and the access control functions that enable the object management layer to interact with the database. Data integration is achieved by the object management and shared repository layers.

5.11.4. THE CASE REPOSITORY

Various components of CASE tools are-

1. **Central Rpository:**
   - The central repository contains the common, integrated and consistent information
   - The central repository acts like a data dictionary
   - It contains product specification, requirement documents, project information and reports.
2. **Upper Case Tools:**
   - Uppercase tool focus on the planning, analysis phase and sometimes the design phase of the software development lifecycle.
3. **Lower case Tools:**
   - Lower CASE software tool that directly supports the implementation and integration tasks.

4. **Integrated CASE Tools**:

- These are type of tools that integrate both upper and lower CASE, for example making it possible to design a form and build the database to support it at the same time.
- An automated system development environment that provides numerous tools to create diagrams, forms and reports.
- It also offers analysis, reporting and code generation facilities and seamlessly shares abd integrates data across and between tools.

5.11.4.1. The Role of the Repository in CASE

The repository for an I-CASE environment is the set of mechanisms and data structures that achieve data/tool and data/data integration. It provides the obvious functions of a database management system, but in addition, the repository performs or precipitates the following functions:

- Data integrity includes functions to validate entries to the repository, ensure consistency among related objects, and automatically perform "cascading" modifications when a change to one object demands some change to objects related to it.
- Information sharing provides a mechanism for sharing information among multiple developers and between multiple tools, manages and controls multiuser access to data and locks or unlocks objects so that changes are not inadvertently overlaid on one another.
- Data/tool integration establishes a data model that can be accessed by all tools in the I-CASE environment, controls access to the data, and performs appropriate configuration management functions.
- Data/data integration is the database management system that relates data objects so that other functions can be achieved.
- Methodology enforcement defines an entity-relationship model stored in the repository that implies a specific paradigm for software engineering
- Document standardization is the definition of objects in the database that leads directly to a standard approach for the creation of software engineering documents.

ANNA UNIVERSITY QUESTION AND ANSWERS

PART A

UNIT V

1. **Highlight the activities in Project Planning. (APR/MAY 2015)**
    - Software scope
    - Resources
    - Project estimation
    - Decomposition
2. **State the importance of scheduling activity in project management. (APR/MAY 2015)**

    Accurate task duration estimates are defined in order to stabilizes customer relations and maintain team morale. With defined task durations, the team knows what to expect and what is expected of them.
3. **Define risk and list its types. (NOV/DEC 2015)**

    Robert Charette presents a conceptual definition of risk: First, risk concerns future happenings. second, that risk involves change, such as in changes of mind, opinion,

actions, or places. . . . [Third,] risk involves choice, and the uncertainty that choice itself entails. Thus paradoxically, risk, like death and taxes, is one of the few certainties of life.

4. **Mr.Koushan is the project manager on a project to build a new cricket stadium in Mumbai, India. After six months of work, the project is 27% complete. At the start of the project, Koushan estimated that it would cost $50,000,000, What is the earned value? (NOV/DEC 2015)**

Budget at completion, BAC. Hence, BAC = 50,000,000

Project Percent Complete = (EV/BAC) * 100
27 /100 = (EV/50,000,000) x 100
27 = EV/50,000,000
EV = 27 X 50,000,000
EV = 1350000000

Estimate at completion (EAC) = BAC/CPI (The estimated total cost at project completion.)
Variance at completion (VAC)= BAC-EAC (The estimated variance between actual total cost and planned total cost at project completion).

5. **Will exhaustive kiting guarantee that the program is 100% correct? (APR/MAY 2016)**

No. There are many times a program runs correctly as designed, but I think there is no such thing as 100% reliability even after very exhaustive testing. Many things within a person's computer can cause a program to not function as designed even if it works for most other users of that program.

6. **What is risk management? (NOV/DEC 2016)**

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

7. **How is productivity and cost related to function points? (NOV/DEC 2016)**

Inconsistent **productivity** rates between projects may be an indication that a standard process is not being followed. Productivity is defined as the ratio of inputs/outputs. For software, productivity is defined as the amount of effort required to deliver a given set of functionality.

The true **cost of software** is the sum of all costs for the life of the project including all expected enhancement and maintenance costs. The more invested up front should reduce per unit cost for future enhancement and maintenance activities. The unit cost can be hours/FP or $/FP.

8. **What are the different types of productivity estimation measures? (APR/MAY 2017)**

- Function Point and Function Point Analysis
- COCOMO
- Cyclomatic Complexity

9. **List two customers related and technology related risks. (APR/MAY 2017)**

Customer related risk

- Have you worked with the customer in the past?
- Does the customer have a solid idea of what is required?
- Technology related risk
- Is the technology to be built new to your organization?
- Do the customer's requirements demand the creation of new algorithms or input or output technology?

10. List out the principles of project scheduling. (NOV/DEC 2017)

-Compartmentalization
- Interdependency
- Time allocation
- Effort validation
- Defined responsibilities
- Defined outcomes
- Defined milestones

11. Write a note on Risk Information Sheet (RIS). (NOV/DEC 2017)

The Risk Information Sheet documents a Risk that may during the life-time of a specific Software Project. Risk Information Sheets can be used in to supplement or in the place of a formal Risk Mitigation, Monitoring and Management (RMMM) Plan.

12. List two advantages of COCOMO model. APR/MAY 2019

- COCOMO Model is used to estimate the project cost
- COCOMO is easy to interpret, predictable and accurate.

13. Compare project risk and Business risk. APR/MAY 2019

- **Project risk** - that the building costs may be higher than expected because of an increase in materials or labor costs.
- **Business risk** - even if the stadium is constructed on time and within budget that it will not make money for the business.

14. What is budgeted cost of work scheduled? NOV/DEC 2019

The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule. During estimation, the work (in person-hours or person-days) of each software engineering task is planned.

BCWS= sum of BCWSi

Hence, BCWSi is the effort planned for work task i.

15. Write any two differences between "known risk" and predictable risk". NOV/DEC 2019

Known risk: It can be uncovered after careful evaluation project plan, business and technical environment in which the project is being developed, other reliable information resources. E.g. unrealistic delivery date, lack of software poor development environment.

Predictable risks are those risks that can be identified in advance based on pas project experience. For example: Experienced and skilled staff leaving the organization in between.

ANNA UNIVERSITY QUESTION AND ANSWERS

PART B

1. State the need for Risk Management and explain the activities under Risk Management. **(APRIL/MAY 2015) (NOV/DEC 2015) (APRIL/MAY 2017)**
2. Write short notes on the following **(APRIL/MAY 2015)**
   (i) Project Scheduling
   (ii) Project Timeline chart and Task network
3. Discuss about COCOMO II model for software estimation. **(NOV/DEC2015)(APRIL/MAY 2017)(NOV/DEC 2019)**
4. Write short notes on the following : (2 x8 = 16) **(APRIL/MAY 2016)**

   (i) Make/Buy decision
   (ii) COCOMO II
5. An application has the following: 10 low external inputs, 8 high external outputs, 13 low internal logical files, 17 high external interface files, 11 average external inquires and complexity adjustment factor of 1.10. What are the unadjusted and adjusted function point counts ? **(APRIL/MAY 2016)**
6. Discuss Putnam resources allocation model. Derive the time and effort equations. **(APRIL/MAY 2016)**
7. Suppose you have a budgeted cost of a project as Rs. 9,00,000. The project is to be completed in 9 months. After a month, you have completed 10 percent of the project at a total expense of Rs. 1,00,000. The planned completion should have been 15 percent. You need to determine whether the project is on-time and on-budget? Use Earned Value analysis approach and interpret. **(NOV/DEC 2016)**

Answer:

Budget at Completion (BAC) = 9,00,000
Actual Cost (AC) = 1,00,000
Planned Completion = 15%
Actual Completion = 10%
Planned Value (PV) = Planned Completion(%) * BAC
= (15/100) * 900000
=1, 35,000

Earned Value (EV) = Actual Completion * BAC
=(10/100) *900000 = 90,000

Cost Performance Index (CPI) = EV/AC
= 90000/100000
CPI = 0.9
Scheduled performance index SPI = EV/PV
= 90000 / 1, 35,000
= 0.66
Therefore, CPI is close to 1 (i.e., 0.9). This means that 90% of work is performed.

But, SPI is less than 1, hence the project team is complexity only 0.66 (approximately 40 mins). So the project is not on time, correction action should be taken.

8. Consider the following Function point components and their complexity. If the total degree of influence is 52, find the estimated function points. **(NOV/DEC 2016)**

Function type	Estimated count	Complexity
ELF	2	7
ILF	4	10
EQ	22	4
EO	16	5
EI	24	4

Answer:

Function type	**Estimated count**	**Complexity**	**Product of Count and Complexity**
External Interface File – ELF	2	7	14
Internal Logical file – ILF	4	10	40
External Inquiries – EQ	22	4	88
External Output - EO	16	5	80
External Inputs – EI	24	4	96
Count Total			**318**

FP = UFP *TCF

TCF = (0.65+0.01*DI)

FP = 318 * (0.65 +0.01 * 52)

FP= 372.06

9. Describe in detail COCOMO model for software cost estimation. Use it to estimate the effort required to build software for a simple ATM that produces 12 screens, 10 reports and has 80 software components. Assume average complexity and average developer maturity. Use application composition model with object points. **(NOV/DEC 2016)**

10. List the features of LOC and FP based estimation models. Compare the two models and list the advantage of over one other. **APR/ MAY 2019**

11. Define risk. List types of risk and explain phases in risk management**. APR/ MAY 2019, NOV/DEC 2019**