**UNIT- I**

**The Software Process**: A generic process model, Process Assessment and Improvement.

**Process Models:** The Waterfall model, Incremental process models, Evolutionary process models, Concurrent Models, The Unified process, Personal and Team Process models.

# Software and Software Engineering

Software engineering stands for the term is made of **two** words, S**oftware** and E**ngineering.**

**Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

**Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.

**Software engineering** is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

***Definitions***

**IEEE** defines software engineering as:

1. The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
2. The study of approaches as in the above statement.

**Fritz Bauer**, a German computer scientist, defines software engineering as:

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

Defining Software Software is defined as 1. Instructions : Programs that when executed provide desired function, features, and performance 2. Data structures : Enable the programs to adequately manipulate information 3. Documents: Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs

\Software Application Domains

Seven Broad Categories of software are challenges for software engineers

System software : A collection of programs written to service other programs. Some system

software (e.g., compilers, editors, and file management utilities)

Application software : Stand-alone programs that solve a specific business need. Application

software is used to control business functions in real time (e.g., point-of-sale transaction

processing, real-time manufacturing process control).

Engineering/scientific software : It has been characterized by “number crunching” algorithms.

Applications range from astronomy to volcanology, from automotive stress analysis to space

shuttle orbital dynamics, and from molecular biology to automated manufacturing.

Embedded software : It 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. Embedded software can perform

limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant

function and control capability (e.g., digital functions in an automobile such as fuel control,

dashboard displays, and braking systems).

Product-line software : Designed to provide a specific capability for use by many different

customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory

control products) or address mass consumer markets (e.g., word processing, spreadsheets,

computer graphics, multimedia, entertainment, database management, and personal and business

financial applications).

Web applications : These Applications called “WebApps,” this network-centric software

category spans a wide array of applications. In their simplest form, WebApps can be little more

than a set of linked hypertext files that present information using text and limited graphics.

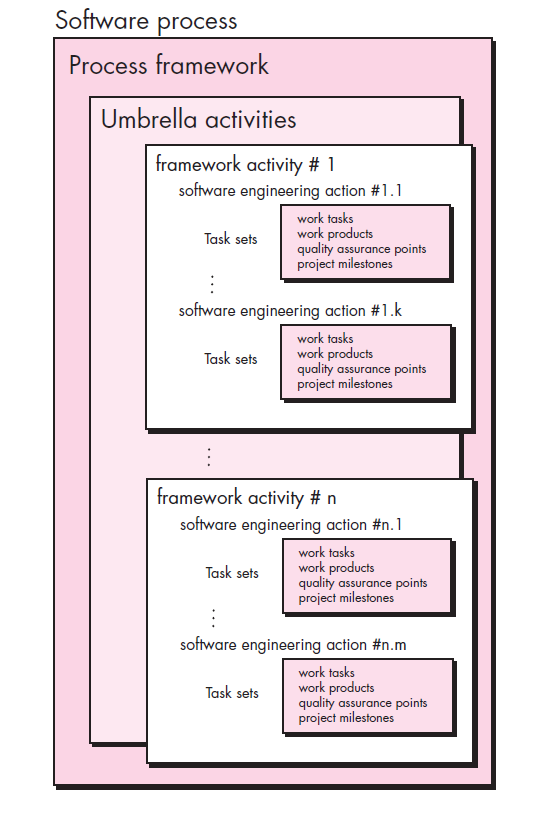
.Artificial intelligence software : These makes use of non numerical algorithms to solve

complex problems that are not amenable to computation or straightforward analysis.

Applications within this area include robotics, expert systems, pattern recognition (image and

voice), artificial neural networks, theorem proving, and game playing

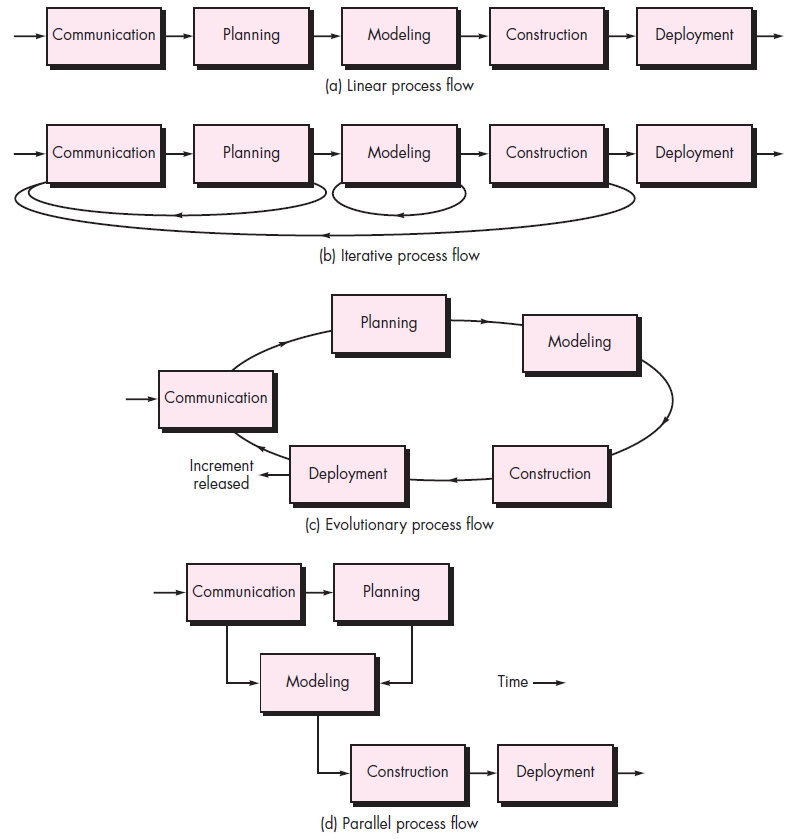
A GENERIC PROCESS MODEL

The software process is represented schematically in following figure. Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

A generic process framework defines **five** framework activities—**communication, planning, modeling, construction,** and **deployment.**

In addition, a set of umbrella activities **project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others** are applied throughout the process.

This aspect is called ***process flow.*** It describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in following figure



A generic process framework for software engineering A ***linear process flow*** executes each of the **five** framework activities in sequence, beginning with communication and culminating with deployment.

An ***iterative process flow*** repeats one or more of the activities before proceeding to the next. An *evolutionary process flow* executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software. A *parallel process flow* executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

#### Defining a Framework Activity

A software team would need significantly more information before it could properly execute any one of these activities as part of the software process. Therefore, you are faced with a key question: What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?

#### Identifying a Task Set

Different projects demand different task sets. The software team chooses the task set based on problem and project characteristics. A task set defines the actual work to be done to accomplish the objectives of a software engineering action.

#### Process Patterns

A ***process pattern*** describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template —**a consistent method for describing problem solutions within the context of the software process**.

Patterns can be defined at any level of abstraction. a pattern might be used to describe a **problem (and solution)** associated with a complete **process model** (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a **framework activity** (e.g., **planning**) or an **action** within a framework activity (e.g., project estimating).

Ambler has proposed a template for describing a process pattern:

**Pattern Name.** The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

**Forces.** The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

**Type.** The pattern type is specified. Ambler suggests **three** types:

* + - * ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **Establishing Communication.** This pattern would incorporate the task pattern **Requirements Gathering** and others.
      * ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., Requirements Gathering is a task pattern).
      * ***Phase pattern***—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **Spira lModel** or **Prototyping.**

**Initial context.** Describes the conditions under which the pattern applies. Prior to the initiation of the pattern:

1. What organizational or team-related activities have already occurred?
2. What is the entry state for the process?
3. What software engineering information or project information already exists?

**Problem.** The specific problem to be solved by the pattern.

**Solution.** Describes how to implement the pattern successfully. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

**Resulting Context.** Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

1. What organizational or team-related activities must have occurred?
2. What is the exit state for the process?
3. What software engineering information or project information has been developed? **Related Patterns.** Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form.

**Known Uses and Examples.** Indicate the specific instances in which the pattern is applicable.

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).

## PROCESS ASSESSMENT AND IMPROVEMENT

Assessment attempts to understand the current state of the software process with the intent of improving it.

A number of different approaches to **software process assessment and improvement** have been proposed over the past few decades.

***Standard CMMI Assessment Method for Process Improvement (SCAMPI)***—provides a **five** step process assessment model that incorporates **five** phases: **initiating, diagnosing, establishing, acting, and learning.** The SCAMPI method uses the SEI CMMI as the basis for assessment.

***CMM-Based Appraisal for Internal Process Improvement (CBA IPI)—*** provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.

**SPICE (ISO/IEC15504)**—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.

**ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

**Software Process**



identifies modifications to

by identifies capabilities and risk of



leads to leads to

Capability Determination

Software Process Improvement

|  |  |  |
| --- | --- | --- |
|  |  | is examined |
| Software Process Assessment | | |



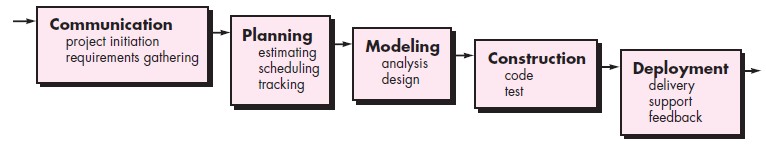
motivates

## PRESCRIPTIVE PROCESS MODELS

Prescriptive process models were originally proposed to bring order to the chaos of software development. Prescriptive process models define a prescribed set of process elements and a predictable process work flow. “prescriptive” 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.

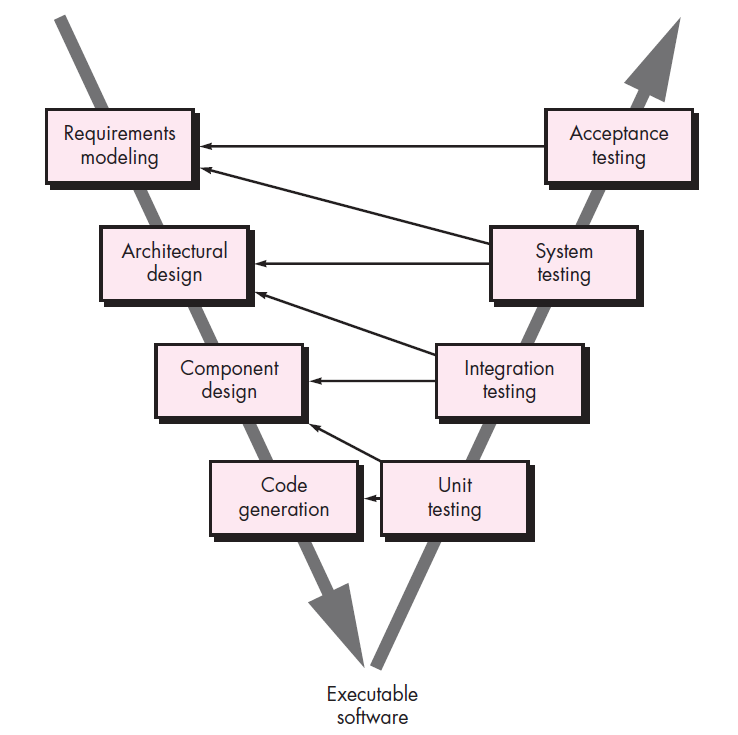
#### The Waterfall Model

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**.





A variation in the representation of the waterfall model is called the ***V-model*.** Represented in following figure. 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 that validate each of the models created as the team moved down the left side. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

The waterfall model is the oldest paradigm for software engineering. The problems that are sometimes encountered when the waterfall model is applied are:

* + - * Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
      * It is often 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.
      * The customer must have patience. A working version of the program(s) will not be available until late in the project time span.

This model is suitable when ever limited number of new development efforts and when requirements are well defined and reasonably stable.

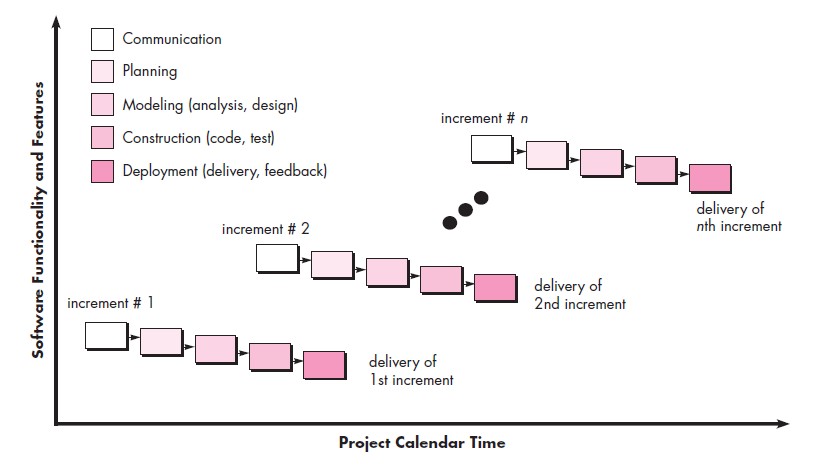
#### Incremental Process Models

The incremental model delivers a series of releases, called increments, that provide progressively more functionality for the customer as each increment is delivered.

The *incremental* model combines elements of linear and parallel process flows discussed in Section 1.7. 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.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

When an incremental model is used, 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 and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated 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.

#### Fig : Incremental Model

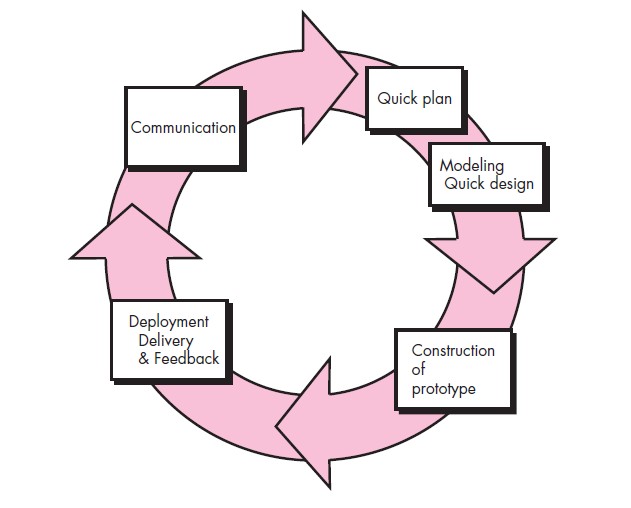
**Evolutionary Process Models**

Evolutionary models are **iterative**. They are characterized in a manner that enables you to develop increasingly more complete versions of the software with each iteration. There are **two** common evolutionary process models.

**Prototyping Model :** Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a ***prototyping paradigm*** may offer the best approach.

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 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.



#### Fig : prototyping paradigm

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.

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 that enable working programs to be generated quickly. The prototype can serve as **“the first system**.” Prototyping can be **problematic** for the following reasons:

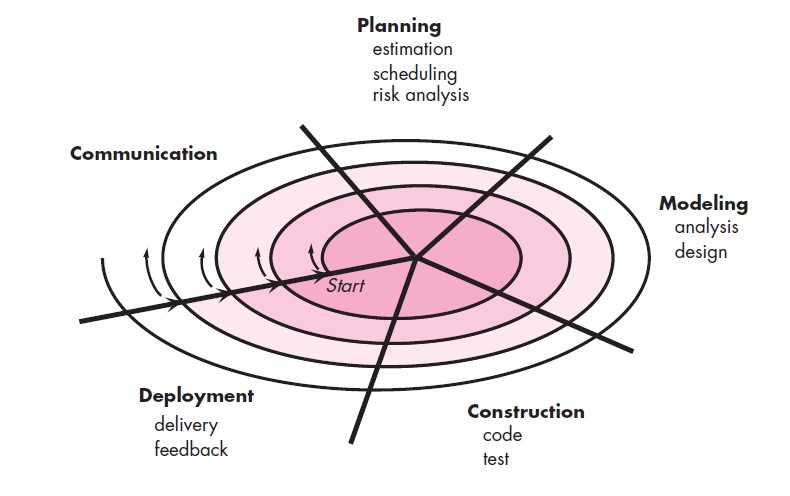
* + - * Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven’t considered overall software quality or long-term maintainability.
      * As a software engineer, you often make implementation compromises in order to get a prototype working quickly. 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.

Although problems can occur, prototyping can be an **effective paradigm** for software engineering.

**The Spiral Model :** Originally proposed by **Barry Boehm**, 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. Boehm describes the model in the following manner

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. One is a ***cyclic approach*** for incrementally growing a system’s degree of definition and implementation while decreasing its degree of risk. The other is a set of ***anchor point milestones*** for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.



#### Fig : The Spiral Model

A spiral model is divided into a set of **framework activities** defined by the software engineering team. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a **clockwise** direction, beginning at the **center**. Risk is considered as each revolution is made. ***Anchor point milestones*** are a combination of work products and conditions that are attained along the path of the spiral are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a **product** specification; subsequent passes around the spiral might be used to develop a **prototype** and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan.

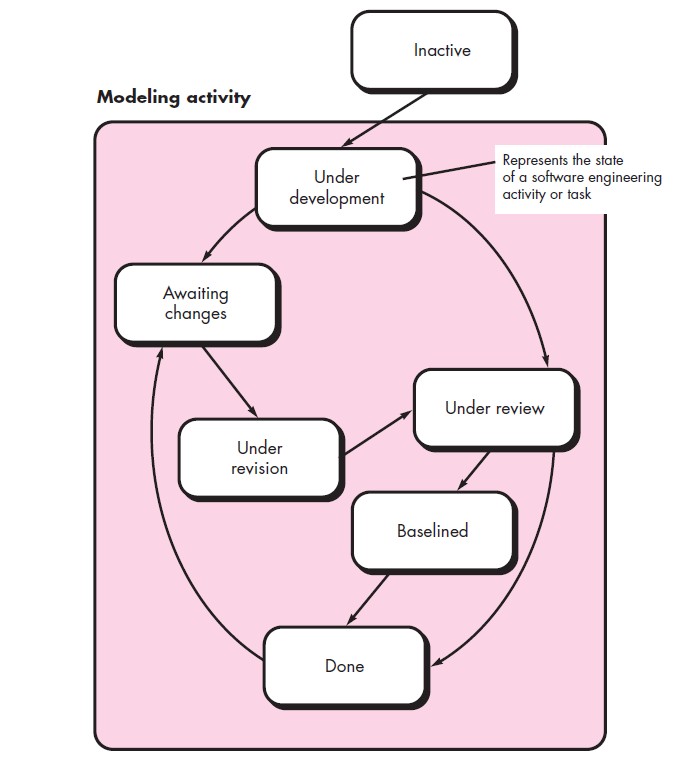
The spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a “**concept development project**” that starts at the core of the spiral and continues for multiple iterations until concept development is complete. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a “**product enhancement project.”**

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. 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.

#### Concurrent 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. The concurrent model is often more appropriate for product engineering projects where different engineering teams are involved.

These models 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.



#### Fig : Concurrent development model

All software engineering activities exist concurrently but reside in different states. 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. 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. 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.

**THE UNIFIED PROCESS**

Unified process (UP) is an architecture-centric, use-case driven, iterative and incremental development process. UP is also referred to as the **unified software development process**.

The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models, but characterize them in a way that implements many of the best principles of **agile software development.** The Unified Process recognizes the importance of **customer communication** and streamlined methods for describing the customer’s view of a system. It emphasizes the important role of software architecture and “helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse” . It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

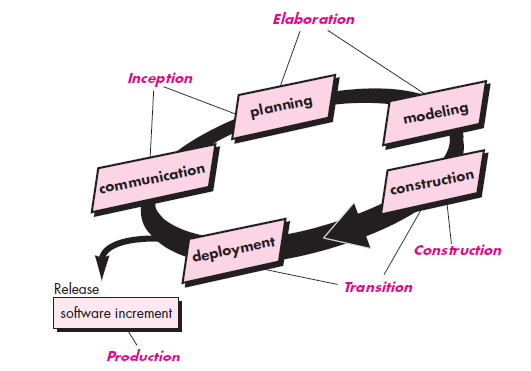
#### A Brief History

During the early 1990s James Rumbaugh, Grady Booch, and Ivar Jacobson began working on a “unified method” that would combine the best features of each of their individual object-oriented analysis and design methods and adopt additional features proposed by other experts in object- oriented modeling. The result was **UML—a *unified modeling language*** that contains a robust notation for the modeling and development of object-oriented systems. They developed the ***Unified Process****,* a framework for object-oriented software engineering using **UML**.

#### Phases of the Unified Process

This process divides the development process into **five** phases:

* Inception
* Elaboration
* Conception
* Transition
* Production



The ***inception phase*** of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.

The ***elaboration phase*** encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include **five different views** of the software—***the use case model, the requirements model, the design model, the implementation model, and the deployment model***. Elaboration creates an “**executable architectural baseline**” that represents a “**first cut**” executable system.

The ***construction phase*** of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment (i.e., the release) are then implemented in **source code**.

The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for **beta testing and user feedback** reports both defects and necessary changes. At the conclusion of the transition phase, the software increment becomes a usable software release.

The ***production phase*** of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated. It is likely that at the same time the construction, transition, and production phases are being conducted, work may have already begun on the next software increment. This means that the **five UP phases** do not occur in a sequence, but rather with staggered concurrency.

## PERSONAL AND TEAM PROCESS MODELS

The best software process is one that is close to the people who will be doing the work. Watts Humphrey proposed two process models. Models - “P**ersonal Software Process (PSP)**” and “**Team Software Process (TSP)**.” Both require hard work, training, and coordination, but both are achievable.

#### Personal Software Process (PSP)

The ***Personal Software Process* (PSP)** emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition **PSP** makes the practitioner responsible for project planning and empowers the practitioner to control the quality of all software work products that are developed. The **PSP** model defines **five** framework activities:

* **Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, defects estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.
* **High-level design.** External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.
* **High-level design review.** Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.
* **Development.** The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.
* **Postmortem.** Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

**PSP** stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners.

#### Team Software Process (TSP)

Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a ***Team Software Process* (TSP).** The goal of TSP is to build a “**self directed**” project team that organizes itself to produce high-quality software.

Humphrey defines the following objectives for TSP:

* Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers.
* Show managers how to coach and motivate their teams and how to help them sustain peak performance.
* Accelerate software process improvement by making CMM23 Level 5 behavior normal and expected.
* Provide improvement guidance to high-maturity organizations.
* Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team’s software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: **project launch, high-level design, implementation, integration and test,** and **postmortem.** TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work. “Scripts” define specific process activities (i.e., project launch, design, implementation, integration and system testing, postmortem) and other more detailed work functions (e.g., development planning, requirements development, software configuration management, unit test) that are part of the team process.

# UNIT-II

# AGILE DEVELOPMENT

## WHAT IS AGILITY?

**Agile** is a software development methodology to build software incrementally using short iterations of 1 to 4 weeks so that the development process is aligned with the changing business needs.

An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

## AGILITY AND THE COST OF CHANGE

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

Agility argue that a well-designed agile process “flattens” the cost of change curve shown in following figure, allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence to suggest that a significant reduction in the cost of change can be achieved.

## AGILE PROCESS

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

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

#### Agility Principles

Agility principles for those who want to achieve agility:

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

tunes and adjusts its behavior accordingly.

#### Human Factors

Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams.” The key point in this statement is that *the process molds to the needs of the people and team*

* **Competence.** In an agile development context, “competence” encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.
* **Common focus.** Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.
* **Collaboration.** Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.
* **Decision-making ability.** Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.
* **Fuzzy problem-solving ability.** Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change.
* **Mutual trust and respect.** The agile team must become what DeMarco and Lister call a “jelled” team. A jelled team exhibits the trust and respect that are necessary to make them “so strongly knit that the whole is greater than the sum of the parts.”
* **Self-organization.** In the context of agile development, self-organization implies **three** things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the workschedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale.

## EXTREME PROGRAMMING (XP)

***Extreme Programming* (XP),** the most widely used approach to agile software development, emphasizes business results first and takes an incremental, get-something-started approach to building the product, using continual testing and revision.

**XP Values**

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

In order to achieve effective ***communication*** between software engineers and other stakeholders, XP emphasizes close, yet informal collaboration between customers and developers, the establishment of effective metaphors3 for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

To achieve ***simplicity****,* XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored* at a later time.

***Feedback*** is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy the software provides the agile team with feedback. XP makes use of the ***unit test*** as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality.

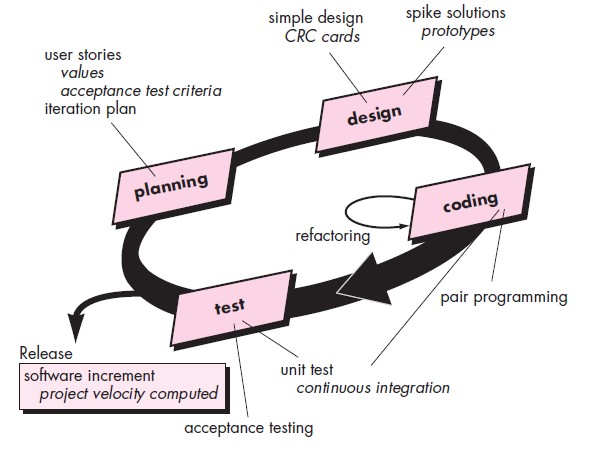
Beck argues that strict adherence to certain XP practices demands ***courage.*** A better word might be ***discipline****.* An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates ***respect*** among it members, between other stakeholders and team members, and indirectly, for the software itself. As they

achieve successful delivery of software increments, the team develops growing respect for the XP process.

#### The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.



Key XP activities are

**Fig : The Extreme Programming process**

* **Planning.** The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.
* **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides

implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a ***spike solution***, the design prototype is implemented and evaluated. XP encourages ***refactoring***—a construction technique that is also a method for design optimization.

Fowler describes r**efactoring** in the following manner: Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [that minimizes the chances of introducing bugs].

* + **Coding.** After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance.

* + **Testing.** The creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages a regression testing strategy whenever code is modified. As the individual unit tests are organized into a “universal testing suite” integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells states: “Fixing small problems every few hours takes less time than fixing huge problems just before the deadline.”

XP ***acceptance tests***, also called ***customer tests****,* are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

#### Industrial XP

Joshua Kerievsky describes *Industrial Extreme Programming* (IXP) in the following manner: “IXP is an organic evolution of XP. It is imbued with XP’s minimalist, customer-centric, test- driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices.” IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

* **Readiness assessment.** Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.* The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.
* **Project community.** Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well- trained, adaptable and skilled, and have the proper temperament to contribute to a self- organizing team. When XP is to be applied for a significant project in a large organization, the concept of the “team” should morph into that of a *community.* A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who “are often at the periphery of an IXP project yet they may play important roles on the project”. In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.
* **Project chartering.** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.
* **Test-driven management.** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable “destinations” and then defines mechanisms for determining whether or not these destinations have been reached.
* **Retrospectives.** An IXP team conducts a specialized technical review after a software increment is delivered. Called a *retrospective,* the review examines “issues, events, and lessons-learned” across a software increment and/or the entire software release. The intent is to improve the IXP process.
* **Continuous learning.** Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product.

## OTHER AGILE PROCESS MODELS

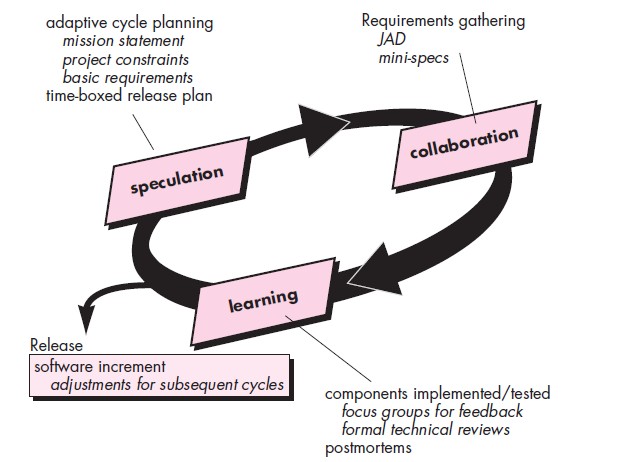
Other agile process models have been proposed and are in use across the industry.

Among the most common are:

* Adaptive Software Development (ASD)
* Scrum
* Dynamic Systems Development Method (DSDM)
* Crystal
* Feature Drive Development (FDD)
* Lean Software Development (LSD)
* Agile Modeling (AM)
* Agile Unified Process (AUP)

#### Adaptive Software Development (ASD)

*Adaptive Software Development* (ASD) has been proposed by Jim Highsmith as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

High smith argues that an agile, adaptive development approach based on collaboration is “as much a source of *order* in our complex interactions as discipline and engineering.” He defines an ASD “life cycle” that incorporates three phases, speculation, collaboration, and learning.

**Fig : Adaptive software development**

During ***speculation****,* the project is initiated and ***adaptive cycle planning*** is conducted. Adaptive cycle planning uses project initiation information—the customer’s mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.

Motivated people use ***collaboration*** in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

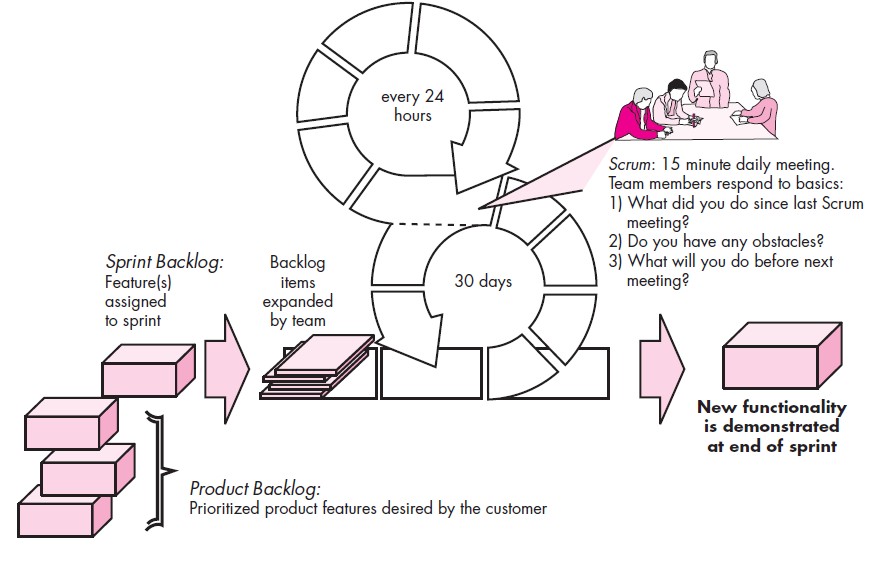
As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on **“learning**” as much as it is on progress toward a completed cycle.

ASD teams learn in **three** ways: **focus groups, technical reviews , and project postmortems**. ASD’s overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

#### Scrum

Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a ***sprint.*** The work conducted within a sprint is adapted to the problem at hand and is defined and often modified

in real time by the Scrum team. The overall flow of the Scrum process is illustrated in following figure

\**Scrum** emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

* ***Backlog***—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time. The product manager assesses the backlog and updates priorities as required.
* ***Sprints***—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.
* ***Scrum meetings***—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members
  + What did you do since the last team meeting?
  + What obstacles are you encountering?
  + What do you plan to accomplish by the next team meeting?

A team leader, called a ***Scrum master****,* leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to “**knowledge socialization**”

* ***Demos***—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

#### Dynamic Systems Development Method (DSDM)

The *Dynamic Systems Development Method* (DSDM) is an agile software development approach that “provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment” The DSDM philosophy is borrowed from a modified version of the **Pareto principle—80 percent of an application can be delivered in 20 percent of the time.** It would take to deliver the complete (100 percent) application. DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to

facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The *DSDM life cycle* that defines **three** different iterative cycles, preceded by **two** additional life cycle activities:

* ***Feasibility study***—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process
* ***Business study***—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.
* ***Functional model iteration****—*produces a set of incremental prototypes that demonstrate functionality for the customer.
* ***Design and build iteration****—*revisits prototypes built during ***functional model iteration*** to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.
* ***Implementation****—*places the latest software increment into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

#### Crystal

Alistair Cockburn and Jim Highsmith created the *Crystal family of agile methods* in order to achieve a software development approach that puts a premium on “maneuverability” during what Cockburn characterizes as “a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game”

The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

#### Feature Driven Development (FDD)

*Feature Driven Development* (FDD) was originally conceived by Peter Coad and his colleagues as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing have extended and improvedCoad’s work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature- based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means.

FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns (for analysis, design, and construction).

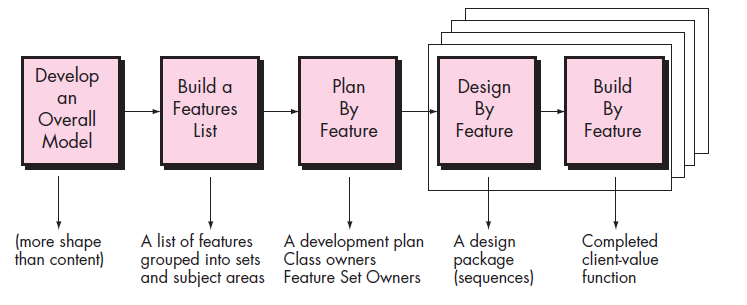
In the context of FDD, a *feature* “is a client-valued function that can be implemented in two weeks or less” The emphasis on the definition of features provides the following benefits:

* Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
* Features can be organized into a hierarchical business-related grouping.
* Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
* Because features are small, their design and code representations are easier to inspect effectively.
* Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues suggest the following template for defining a feature:

#### <action> the <result> <by for of to> a(n) <object>

where an **<object>** is “a person, place, or thing



#### Fig : Feature Driven Development (FDD)

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. FDD defines **six** milestones during the design and implementation of a feature: “**design walkthrough, design, design inspection, code, code inspection, promote to build**”

#### Lean Software Development (LSD)

*Lean Software Development* (LSD) has adapted the principles of lean manufacturing to the world of software engineering. The lean principles that inspire the LSD process can be summarized as ***eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people,* and *optimize the whole****.* Each of these principles can be adapted to the software process.

#### Agile Modeling (AM)

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don’t have to be perfect.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility

to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Agile Modeling suggests a wide array of “core” and “supplementary” modeling principles, those that make AM unique are :

* **Model with a purpose.** A developer who uses AM should have a specific goal in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.
* **Use multiple models.** There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.
* **Travel light.** As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler notes that “Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner
* **Content is more important than representation.** Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.
* **Know the models and the tools you use to create them.** Understand the strengths and weaknesses of each model and the tools that are used to create it.
* **Adapt locally.** The modeling approach should be adapted to the needs of the agile team.

#### Agile Unified Process (AUP)

The ***Agile Unified Process*** (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building computer-based systems. By adopting the classic UP phased activities— *inception, elaboration, construction,* and *transition*—AUP provides a serial overlay that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities.

* ***Modeling****.* UML representations of the business and problem domains are created.
* ***Implementation.*** Models are translated into source code.
* ***Testing****.* Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
* ***Deployment.*** Like the generic process activity deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
* ***Configuration and project management.*** In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.
* ***Environment management.*** Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

# Understanding Requirements

## REQUIREMENTS ENGINEERING

**Requirements** analysis, also called **requirements engineering**, is the process of determining user expectations for a new or modified product. Requirements engineering is a major software engineering action that begins during the **communication activity and continues into the modeling activity.** It must be adapted to the needs of the process, the project, the product, and the people doing the work. Requirements engineering builds a bridge to design and construction.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses **seven** distinct tasks: **inception, elicitation, elaboration, negotiation, specification, validation, and management**.

**Inception :** It establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

**Elicitation:** In this stage, proper information is extracted to prepare to document the requirements. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product

fits into the needs of the business, and finally, how the system or product is to be used on a day- to-day basis.

* + - **Problems of scope.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
    - **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don’t have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be “obvious,” specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or un testable.
    - **Problems of volatility.** The requirements change over time.

**Elaboration:** The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information. Elaboration is driven by the creation and refinement of user scenarios that describe **how** the end user (and other actors) will interact with the system.

**Negotiation:** To negotiate the requirements of a system to be developed, it is necessary to identify conflicts and to resolve those conflicts. You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

**Specification:** The term *specification* means **different things to different people**. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

**Validation:** The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies,

omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the **technical review**. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic requirements.

**Requirements management.** Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques.

**ELICITING REQUIREMENTS**

Requirements elicitation (also called *requirements gathering*) combines elements of problem solving, elaboration, negotiation, and specification

#### Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

* + - * Meetings are conducted and attended by both software engineers and other stakeholders.
      * Rules for preparation and participation are established.
      * An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
      * A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
      * A “definition mechanism” (can be work sheets, flip charts, or wall stickers or
      * an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

During inception basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a **one- or two-page “product request**.”

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person’s perception of the system.

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything.

#### Quality Function Deployment

*Quality function deployment* (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD “**concentrates on maximizing customer satisfaction from the software engineering process**”. To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

QFD identifies **three** types of requirements :

* + - * **Normal requirements.** The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.
      * **Expected requirements.** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.
      * **Exciting requirements.** These features go beyond the customer’s expectations and prove to be very satisfying when present.

Although QFD concepts can be applied across the entire software process, QFD uses customer interviews and observation, surveys, and examination of historical data as raw data for the requirements gathering activity. These data are then translated into a table of requirements— called the ***customer voice table***—that is reviewed with the customer and other stakeholders.

#### Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called ***use cases***, provide a description of how the system will be used.

#### Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

* + - * A statement of need and feasibility.
      * A bounded statement of scope for the system or product.
      * A list of customers, users, and other stakeholders who participated in requirements elicitation.
      * A description of the system’s technical environment.
      * A list of requirements and the domain constraints that apply to each.
      * A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
      * Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

# DEVELOPING USE CASES

Use cases are defined from an actor’s point of view. An actor is a role that people (users) or devices play as they interact with the software.

The first step in writing a use case is to define the set of “actors” that will be involved in the story. *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.

Actors represent the roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system. It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. Different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify **primary actors** during the first iteration and **secondary actors** as more is learned about the system.

*Primary actors* interact to achieve required system function and derive the intended benefit from the system. *Secondary actors* support the system so that primary actors can do their work. Once actors have been identified, use cases can be developed.

Jacobson suggests a number of questions that should be answered by a use case:

* Who is the primary actor, the secondary actor(s)?
* What are the actor’s goals?
* What preconditions should exist before the story begins?
* What main tasks or functions are performed by the actor?
* What exceptions might be considered as the story is described?
* What variations in the actor’s interaction are possible?
* What system information will the actor acquire, produce, or change?
* Will the actor have to inform the system about changes in the external environment?
* What information does the actor desire from the system?
* Does the actor wish to be informed about unexpected changes?

The basic use case presents a high-level story that describes the interaction between the actor and the system.

# BUILDING THE REQUIREMENTS MODEL

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require..

#### Elements of the Requirements Model

The specific elements of the requirements model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most requirements models.

* + - * **Scenario-based elements.** The system is described from the user’s point of view using a scenario-based approach.
      * **Class-based elements.** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.
      * **Behavioral elements.** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.
      * **Flow-oriented elements.** Information is transformed as it flows through a computer- based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms.

#### Analysis Patterns

*Analysis patterns* suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler suggest two benefits that can be associated with the use of analysis patterns:

**First**, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations.

**Second**, analysis patterns facilitate the transformation *of* the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the analysis model by reference to the pattern name.

# NEGOTIATING REQUIREMENTS

The intent of negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team. The best negotiations strive for a “**win-win**” result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you win by working to realistic and achievable budgets and deadlines.

Boehm defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem’s key stakeholders.
2. Determination of the stakeholders’ “win conditions.”
3. Negotiation of the stakeholders’ win conditions to reconcile them into a set of win-win conditions for all concerned.

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

# VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments.

A review of the requirements model addresses the following questions:

* + - Is each requirement consistent with the overall objectives for the system/product?
    - Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
    - Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
    - Is each requirement bounded and unambiguous?
    - Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
    - Do any requirements conflict with other requirements?
    - Is each requirement achievable in the technical environment that will house the system or product?
    - Is each requirement testable, once implemented?
    - Does the requirements model properly reflect the information, function, and behavior of the system to be built?
    - Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system?
    - Have requirements patterns been used to simplify the requirements model?
    - Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

UNIT-III

**Requirements Modeling**: Requirements modelling approaches, Scenario based modeling, Data Modeling Concepts, Class-based modeling, Flow-oriented modeling, Creating a behavioral model.

**Design Concepts**: The Design process, Design concepts, The Design model.

#### Requirements Modeling Approaches

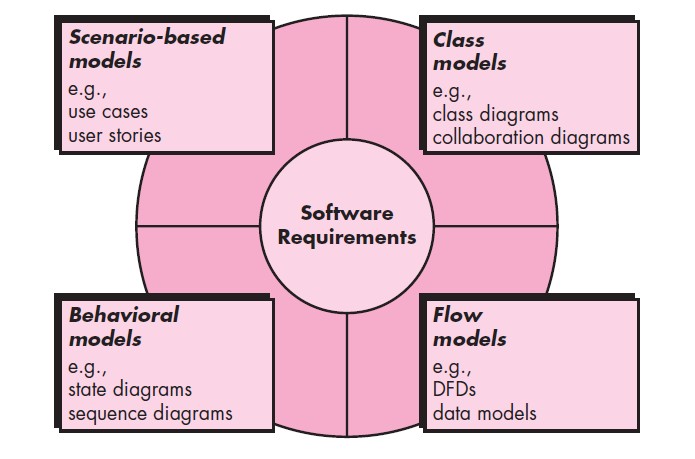
One view of requirements modeling, called ***structured analysis****,* considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their *attributes and relationships*.

A second approach to analysis modeling, called ***object-oriented analysis****,* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Each element of the requirements model is represented in following figure presents the problem from a different point of view.

**Scenario-based elements** depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

**Class-based elements model** the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined.



#### Fig : Elements of the analysis model

**Behavioral elements** depict how external events change the state of the system or the classes that reside within it. Finally,

**Flow-oriented elements** represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

# SCENARIO-BASED MODELING

**Scenario-based elements** depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

#### Creating a Preliminary Use Case

Alistair Cockburn characterizes a use case as a “contract for behavior”, the “contract” defines the way in which an actor uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself.

A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. These are the questions that must be answered if use cases are to provide value as a requirements modeling tool. (1) what to write about, (2) how much to write about it,

(3) how detailed to make your description, and (4) how to organize the description?

To begin developing a set of use cases, list the functions or activities performed by a specific actor*.*

#### Refining a Preliminary Use Case

Each step in the primary scenario is evaluated by asking the following questions:*Can the actor take some other action at this point?*

* + - * *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
      * *Is it possible that the actor will encounter some other behavior at this point (e.g.,behavior that is invoked by some event outside the actor’s control)?* If so, what might it be?

Cockburn recommends using a “brainstorming” session to derive a reasonably complete set of exceptions for each use case. In addition to the **three** generic questions suggested earlier in this section, the following issues should also be explored:

* + - * *Are there cases in which some “validation function” occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
      * *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
      * *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

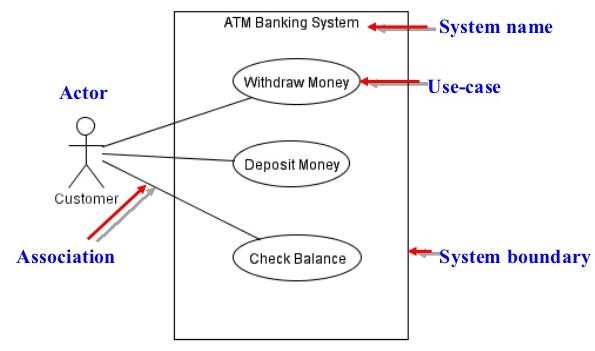
#### Writing a Formal Use Case

The typical outline for formal use cases can be in following manner

* + - * The ***goal in context*** identifies the overall scope of the use case.
      * The ***precondition*** describes what is known to be true before the use case is initiated.
      * The ***trigger*** identifies the event or condition that “gets the use case started”
      * The ***scenario*** lists the specific actions that are required by the actor and the appropriate system responses.
      * ***Exceptions*** identify the situations uncovered as the preliminary use case is refined Additional headings may or may not be included and are reasonably self-explanatory.

Every modeling notation has limitations, and the use case is no exception. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer.

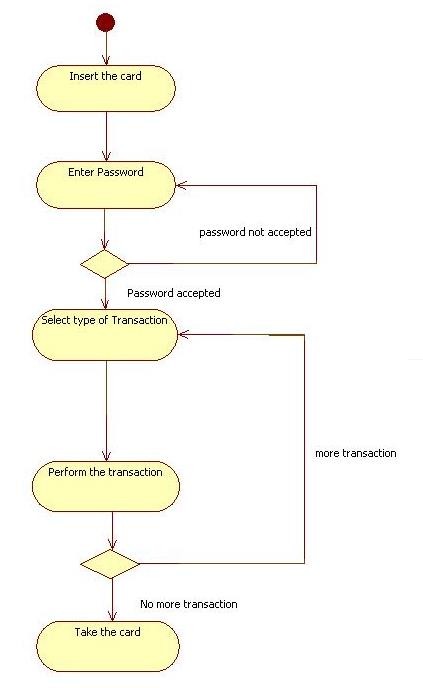


**Fig : Simple Use Case Diagram**

**UML MODELS THAT SUPPLEMENT THE USE CASE**

#### Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. i.e A UML activity diagram represents the actions and decisions that occur as some function is performed.

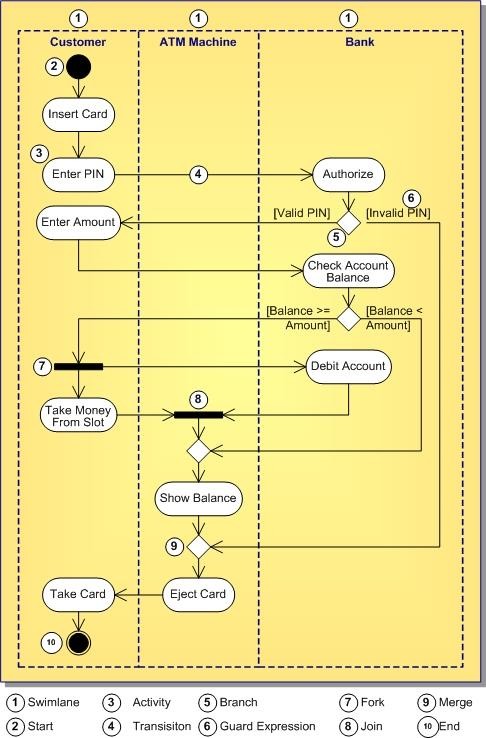


#### Fig : Activity Diagram for ATM

**Swimlane Diagrams**

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

The following figure represents *swimlane diagram for ATM*



**Fig : *swimlane diagram for ATM***

## DATA MODELING CONCEPTS

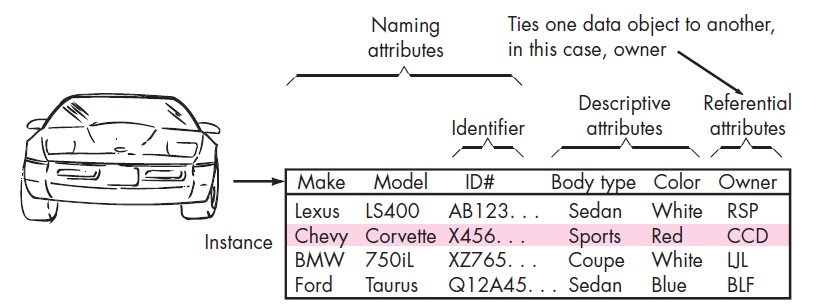
Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way [data](http://searchdatamanagement.techtarget.com/definition/data) needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application. The most widely used data Model by the Software engineers is **E*ntity- Relationship Diagram* (ERD)**, it addresses the issues and represents all data objects that are entered, stored, transformed, and produced within an application.

#### Data Objects

A *data object* is a representation of composite information that must be understood by software. A data object can be an **external entity** (e.g., anything that produces or consumes information), **a thing** (e.g., a report or a display), **an occurrence** (e.g., a telephone call) or **event** (e.g., an alarm), **a role** (e.g., salesperson), **an organizational unit** (e.g., accounting department), **a place** (e.g., a warehouse), **or a structure** (e.g., a file).

For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table as shown in following table. The headings in the table reflect attributes of the object.



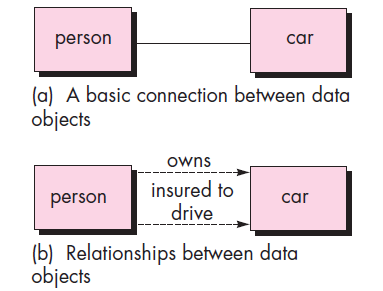
#### Fig : Tabular representation of data objects

**Data Attributes**

*Data attributes* define the properties of a data object and take on one of **three** different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table.

#### Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car.** These objects can be represented using the following simple notation and relationships are 1) A person *owns* a car, 2) A person *is insured to drive* a car



**Fig : Relationships between data objects**

## CLASS-BASED MODELING

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

#### Identifying Analysis Classes

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a “**grammatical parse**” on the use cases developed for the system to be built.

*Analysis classes* manifest themselves in one of the following ways:

* + - * ***External entities*** (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
      * ***Things*** (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
      * ***Occurrences or events*** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
      * ***Roles*** (e.g., manager, engineer, salesperson) played by people who interact with the system.
      * ***Organizational units*** (e.g., division, group, team) that are relevant to an application.
      * ***Places*** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
      * ***Structures*** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

Coad and Yourdon suggest **six** selection characteristics that should be used as you consider each potential class for inclusion in the **analysis model**:

1. ***Retained information****.* The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. ***Needed services****.* The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. ***Multiple attributes****.* During requirement analysis, the focus should be on “major” information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. ***Common attributes****.* A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. ***Common operations****.* A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. ***Essential requirements****.* External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

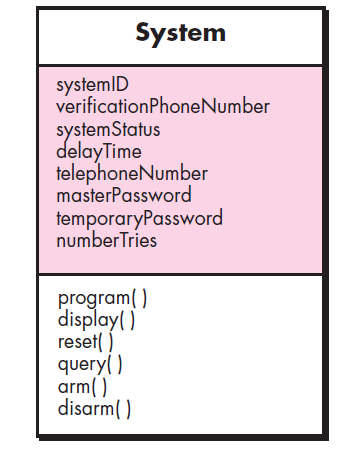
#### Specifying Attributes

*Attributes* describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those “things” that reasonably “belong” to the class.

#### Defining Operations

*Operations* define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event.



#### Fig : Class diagram for the system class

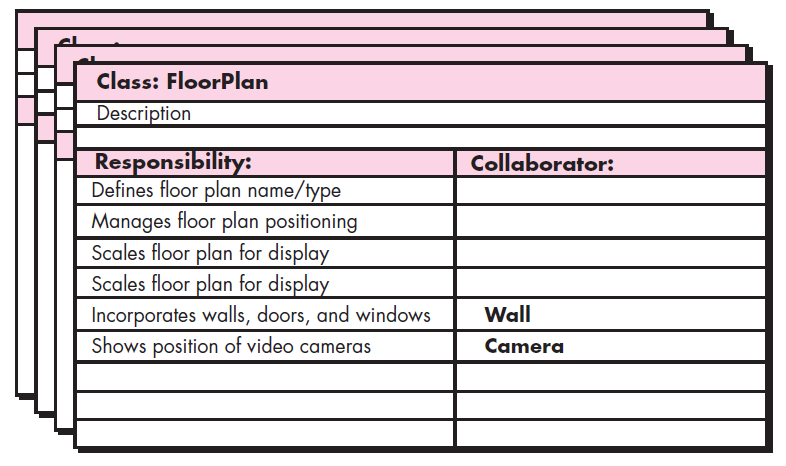
**Class-Responsibility-Collaborator (CRC) Modeling**

*Class-responsibility-collaborator (CRC) modeling* provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.

Ambler describes CRC modeling in the following way :

A CRC model is really a collection of standard **index cards** that represent classes. The cards are divided into **three** sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the **left** and the collaborators on the **right**.

The CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. ***Responsibilities*** are the attributes and operations that are relevant for the class. i.e., a responsibility is “anything the class knows or does” ***Collaborators*** are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a *collaboration* implies either a request for information or a request for some action. A simple CRC index card is illustrated in following figure.



#### Fig : A CRC model index card

**Classes :** The taxonomy of class types can be extended by considering the following categories:

* ***Entity classes***, also called ***model* or *business*** classes, are extracted directly from the statement of the problem. These classes typically represent things that are to be stored in a database and persist throughout the duration of the application.
* ***Boundary classes*** are used to create the interface that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
* ***Controller classes*** manage a “unit of work” from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

**Responsibilities :** Wirfs-Brock and her colleagues suggest five guidelines for allocating responsibilities to classes:

* + - 1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do.
      2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities should reside high in the class hierarchy
      3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
      4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
      5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time.

**Collaborations.** Classes fulfill their responsibilities in one of **two** ways:

1. A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
2. A class can collaborate with other classes.

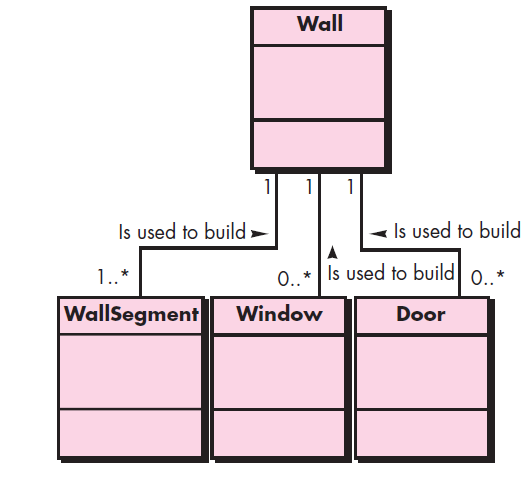
When a complete CRC model has been developed, stakeholders can review the model using the following approach :

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
4. When the token is passed, the holder of the card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

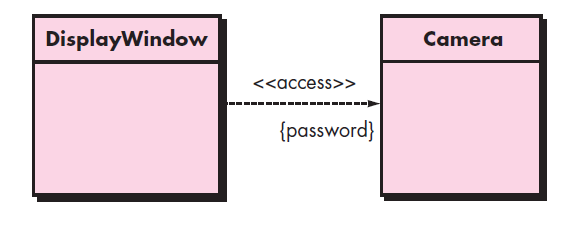
#### Associations and Dependencies

An ***association*** defines a relationship between classes. An association may be further defined by indicating ***multiplicit****y.* **Multiplicity** defines how many of one class are related to how many of another class.

A client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a ***dependency relationship*** is established. Dependencies are defined by a **stereotype**. A ***stereotype*** is an “**extensibility mechanism**” within UML that allows you to define a special modeling element whose semantics are custom defined. In UML. Stereotypes are represented in double angle brackets (e.g., **<<stereotype>>**).



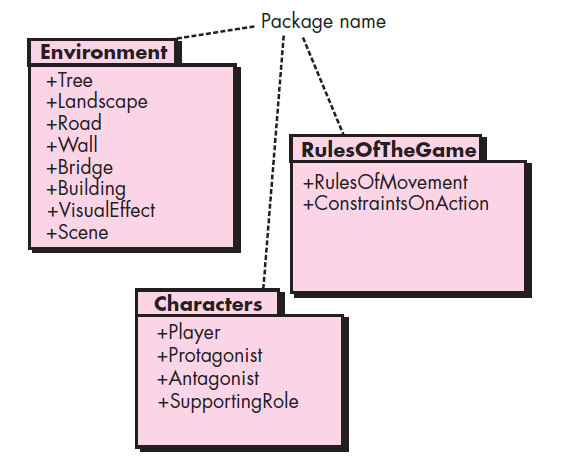
#### Fig : Multiplicity



**Fig : Dependencies**

#### Analysis Packages

An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.



**Fig : Packages**

## Requirements Modeling (Flow, Behavior, Patterns and WEBAPPS)

**REQUIREMENTS MODELING STRATEGIES**

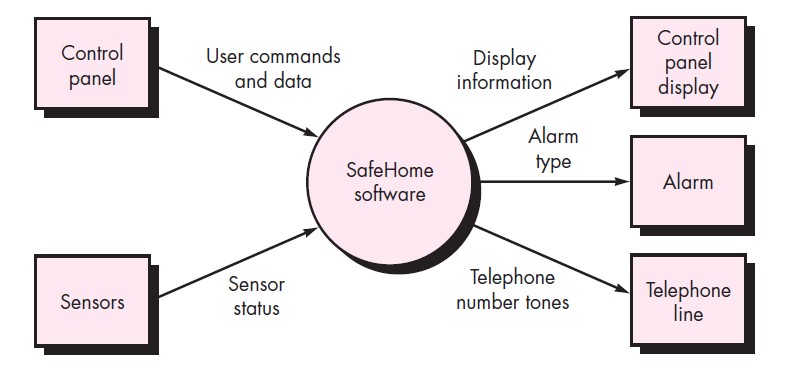
One view of requirements modeling, called ***structured analysis****,*. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeled, called ***object-oriented analysis****,* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.

## FLOW-ORIENTED MODELING

Flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today. The ***data flow diagram* (DFD)** is the representation of Flow-oriented modeling. **The purpose of data flow diagrams is to provide a semantic bridge between users and systems developers.”**

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled **arrows**, and transformations are represented by **circles (also called bubbles).** The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level **0 DFD or *context diagram***) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.



#### Fig : Context-level DFD for the Safe Home security function

**Creating a Data Flow Model**

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram:

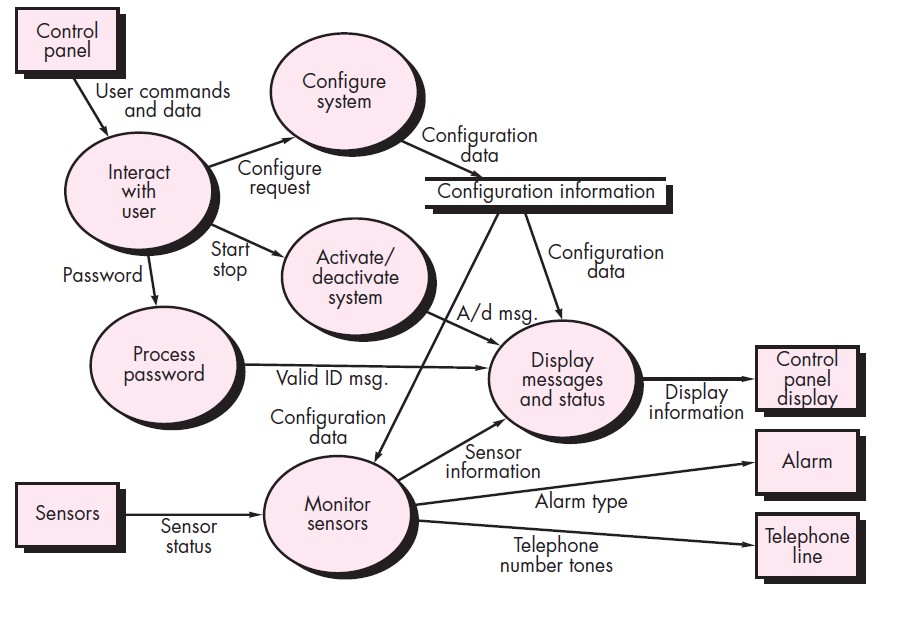
* + - 1. The level 0 data flow diagram should depict the software/system as a single bubble;
      2. Primary input and output should be carefully noted;
      3. Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level;
      4. All arrows and bubbles should be labeled with meaningful names;
      5. *Information flow continuity* must be maintained from level to level,2 and
      6. One bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram.

A **level 0** DFD for the security function is shown in above figure. The primary ***external entities* (boxes)** produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies.

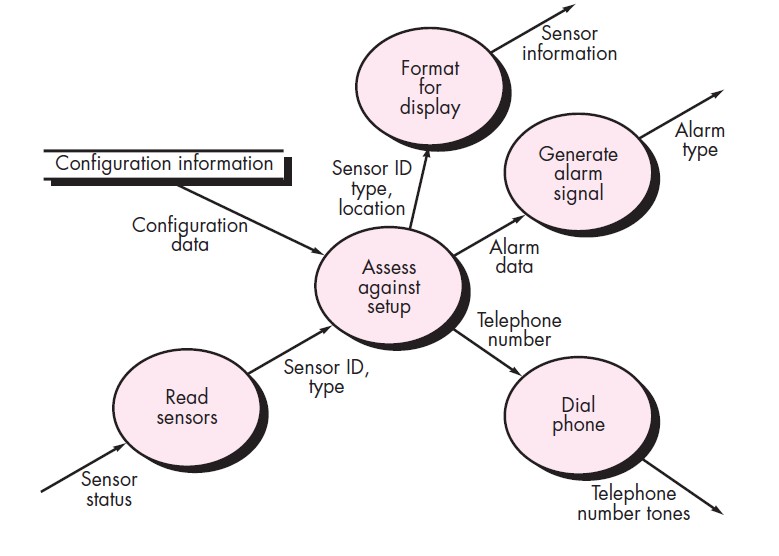
The **level 0** DFD must now be expanded into a **level 1** data flow model. you should apply a “grammatical parse” to the use case narrative that describes the context-level bubble. That is, isolate all nouns (and noun phrases) and verbs (and verb phrases). *The grammatical parse is not*

*foolproof, but it can provide you with an excellent jump start, if you’re struggling to define data objects and the transforms that operate on them.*

The processes represented at **DFD level 1** can be further refined into **lower levels**. The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. a concept, C***ohesion*** can be used to assess the processing focus of a given function. i.e refine DFDs until each bubble is “**single-minded**.”



#### Fig: Level 1 DFD for SafeHome security function



**Fig : Level 2 DFD that refines the monitor sensors process**

#### Creating a Control Flow Model

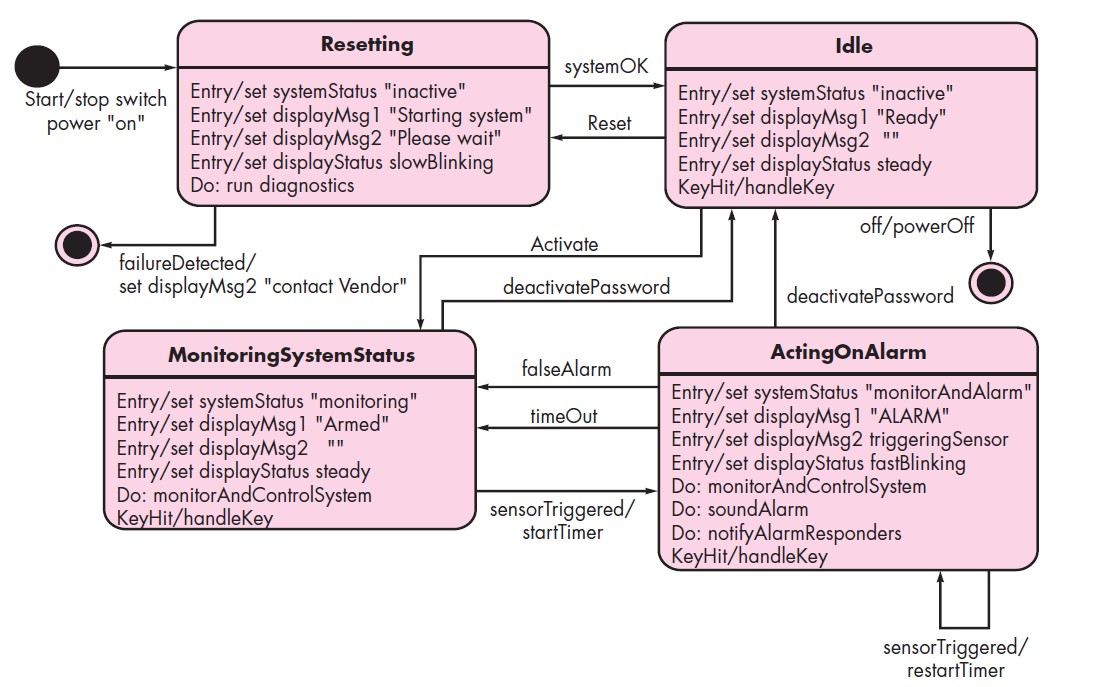
The data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. The following guidelines are suggested for creating a Control Flow Model

* List all sensors that are “read” by the software.
* List all interrupt conditions.
* List all “switches” that are actuated by an operator.
* List all data conditions.
* Recalling the noun/verb parse that was applied to the processing narrative, review all “control items” as possible control specification inputs/outputs.
* Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states.
* Focus on possible omissions—a very common error in specifying control;

#### The Control Specification

A *control specification* (CSPEC) represents the behavior of the system in **two** different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior. The following figure depicts a preliminary state diagram for the level 1 control flow model for *SafeHome.* The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, we can determine the behavior of the system and, more important, ascertain whether there are “holes” in the specified behavior.

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior.



#### Fig : State diagram for SafeHome security function

**The Process Specification**

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm,

mathematical equations, tables, or UML activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, you can create a “mini-spec” that serves as a guide for design of the software component that will implement the bubble.

## CREATING A BEHAVIORAL MODEL

The *behavioral model* indicates how software will respond to external events or stimuli.

To create the model, you should perform the following steps:

* + 1. Evaluate all use cases to fully understand the sequence of interaction within the system.
    2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
    3. Create a sequence for each use case.
    4. Build a state diagram for the system.
    5. Review the behavioral model to verify accuracy and consistency.

#### Identifying Events with the Use Case

The use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. A use case is examined for points of information exchange. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

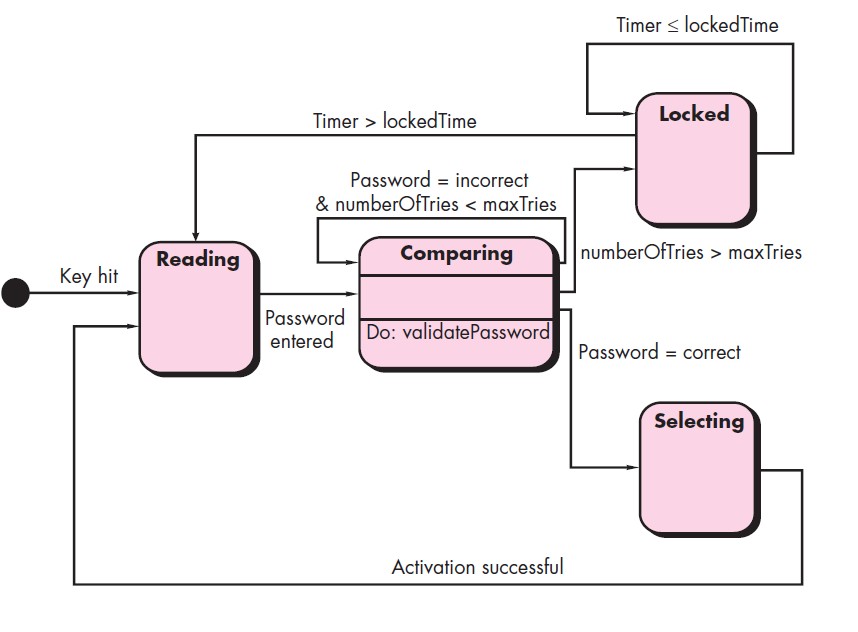
The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed. Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events .

#### State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its Function **Two** different behavioral representations are discussed in the paragraphs that follow. The **first** indicates how

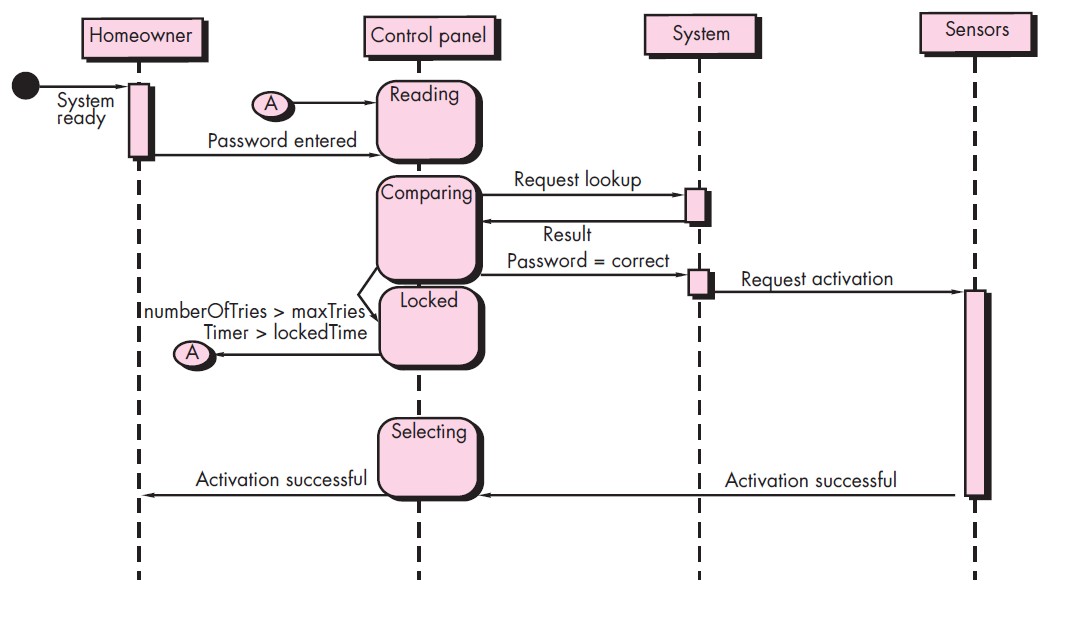
an individual class changes state based on external events and the **second** shows the behavior of the software as a function of time.

**State diagrams for analysis classes.** One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states. The following figure illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function. Each arrow shown in figure represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition



#### Fig : State diagram for the Control Panel class

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



**Fig : Sequence diagram (partial) for the *SafeHome* security function**

## PATTERNS FOR REQUIREMENTS MODELING

Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered. In some cases, the domain knowledge is applied to a new problem within the same application domain. The domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.

The pattern can be reused when performing requirements modeling for an application within a domain. Analysis patterns are stored in a repository so that members of the software team can use search facilities to find and reuse them. Once an appropriate pattern is selected, it is integrated into the requirements model by reference to the pattern name.

#### Discovering Analysis Patterns

The requirements model is comprised of a wide variety of elements: **scenario-based (use cases), data-oriented (the data model), class-based, flow-oriented, and behavioral**. Each of these elements examines the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains.

The most basic element in the description of a requirements model is the **use case**. Use cases may serve as the basis for discovering one or more analysis patterns.

A ***semantic analysis pattern* (**SAP) “is a pattern that describes a small set of coherent use cases that together describe a basic generic application”

### THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a “**blueprint**” for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a **high level of abstraction**

#### Software Quality Guidelines and Attributes

McGlaughlin suggests **three** characteristics that serve as a guide for the evaluation of a good design:

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

**Quality Guidelines.** In order to evaluate the quality of a design representation, consider the following guidelines:

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion,2 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.** Hewlett-Packard developed 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 ease with which a system can be installed, and the ease with which problems can be localized.

#### The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner. Procedural aspects of design definition evolved into a philosophy called ***structured programming****.*

A number of design methods, growing out of the work just noted, are being applied throughout the industry. All of these methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

### DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

* What criteria can be used to partition software into individual components?
* How is function or data structure detail separated from a conceptual representation of the software?
* What uniform criteria define the technical quality of a software design?

The following brief overview of important software design concepts that span both traditional and object-oriented software development.

#### Abstraction

*Abstraction* is the act of representing essential features without including the background details or explanations. the *abstraction* is used to reduce complexity and allow efficient design and implementation of complex *software* systems. Many levels of abstraction can be posed. At the **highest level** of abstraction, a solution is stated in broad terms using the language of the problem environment. At **lower levels** of abstraction, a more detailed description of the solution is provided.

As different levels of abstraction are developed, you work to create both **procedural** and

#### data abstractions.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

A ***data abstraction*** is a named collection of data that describes a data object.

#### Architecture

***Software architecture*** alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” 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.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

* **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.
* **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
* **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

The architectural design can be represented using one or more of a number of different models.

***Structural models:*** *R*epresent architecture as an organized collection of program components. ***Framework models:*** Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

***Dynamic models*** *:* Address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

***Process models*** *:*Focus on the design of the business or technical process that the system must accommodate.

***Functional models*** can be used to represent the functional hierarchy of a system.

A number of different ***architectural description languages* (ADLs)** have been developed to represent these models.

#### Patterns

Brad Appleton defines a ***design pattern*** in the following manner: “A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns”

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.

#### Separation of Concerns

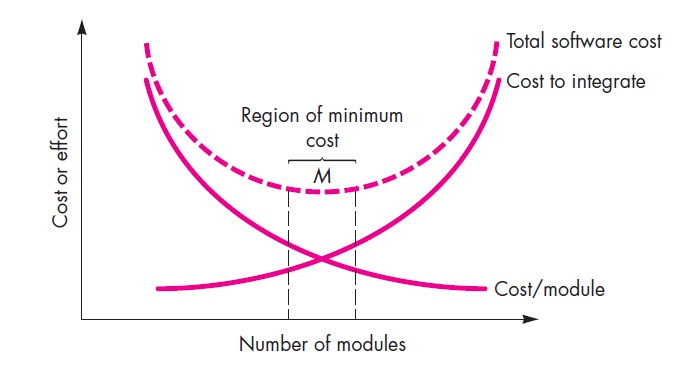
***Separation of concerns*** is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

#### Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called ***module.***

Modularity is the single attribute of software that allows a program to be intellectually manageable



#### Fig : Modularity and software cost

**Information Hiding**

The principle of information hiding suggests that modules be “characterized by design decisions that hides from all others.” In other words, modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

#### Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with “**single minded**” function and an “aversion” to excessive interaction with other modules.

Independence is assessed using **two** qualitative criteria: **cohesion** and **coupling**.

***Cohesion*** is an indication of the relative functional strength of a module. ***Coupling*** is an indication of the relative interdependence among modules.

**Cohesion** is 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. Stated simply, a cohesive module should do just one thing. Although you should always strive for **high cohesion** (i.e., single-mindedness).

**Coupling** is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the **lowest possible coupling**.

#### Refinement

Stepwise refinement is a **top-down** design strategy originally proposed by Niklaus Wirth. Refinement is actually a process of ***elaboration****.* You begin with a statement of function that is defined at a high level of abstraction.

**Abstraction** and **refinement** are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses.

#### Aspects

An ***aspect*** is a representation of a crosscutting concern. A crosscutting concern is some characteristic of the system that applies across many different requirements.

#### Refactoring

An important design activity suggested for many agile methods, ***refactoring*** is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. **Fowler** defines refactoring in the following manner: “**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.”**

#### 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, among others.

#### Design Classes

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. 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, each representing a different layer of the design architecture, can be developed:

* + - * ***User interface classes*** define all abstractions that are necessary for human computer interaction (HCI). The design classes for the interface may be visual representations of the elements of the metaphor.
      * ***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.
      * ***Process classes*** implement lower-level business abstractions required to fully manage the business domain classes.
      * ***Persistent classes*** represent data stores (e.g., a database) that will persist beyond the execution of the software.
      * ***System classes*** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is “**well- formed**.” They define **four** characteristics of a well-formed design class:

* **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. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.
* **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.
* **High cohesion.** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
* **Low coupling.** Within the design model, it is necessary for design classes to collaborate with one another. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain over time.

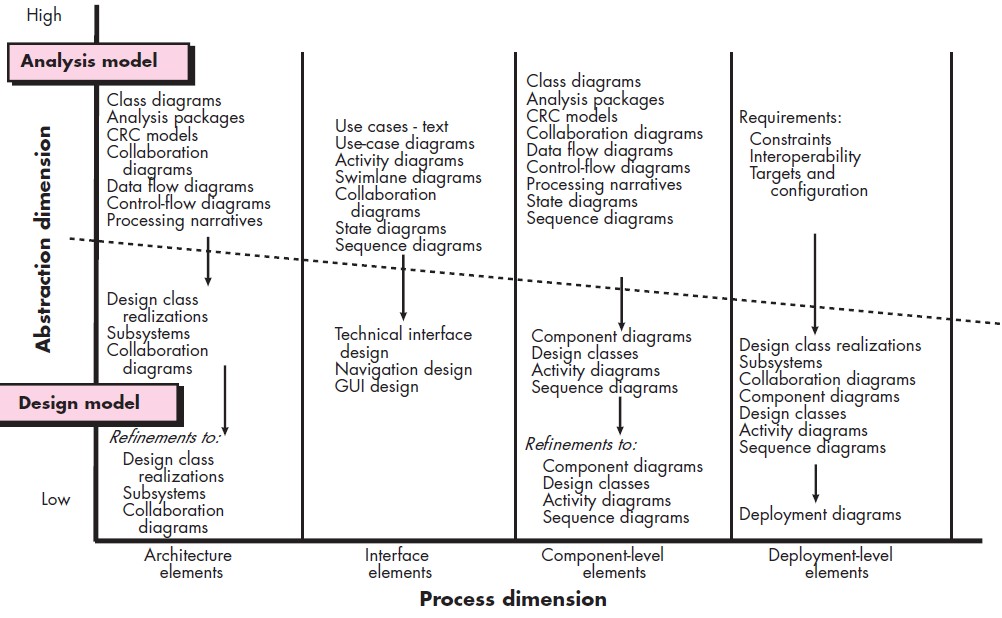
### THE DESIGN MODEL

The design model can be viewed in **two** different dimensions. The ***process dimension*** indicates the evolution of the design model as design tasks are executed as part of the software process. The ***abstraction dimension*** represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively. The design model has **four** major elements: data, architecture, components, and interface.

#### Data Design Elements

Data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user’s view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. The structure of data has always been an important part of software design. At the program **component level**, the 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**, the 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.



#### Fig : Dimensions of the design model

**Architectural Design Elements**

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

The architectural model is derived from **three** sources: (1) information about the application domain for the software to be built; (2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and (3) the availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.

#### Interface Design Elements

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

There are **three** important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components.

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

#### Component-Level Design Elements

The component-level design for software is the equivalent to a set of detailed drawings for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, sinks, showers, tubs, drains, cabinets, and closets.

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

#### 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. Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used and elements of the configuration are explicitly described.

**UNIT- IV**

**Architectural Design**: Software architecture, Architectural styles, Architectural design, Assessing Alternative Architectural Designs, Architectural Mapping using Data flow

**Component-level Design**: What is a component?, Designing class-based components, Conducting component-level design, Designing traditional components, Component-Based Development

### SOFTWARE ARCHITECTURE

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components. It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

#### What Is Architecture?

Bass, Clements, and Kazman define this elusive term in the following way:

#### “The software architecture of a program or computing system is the structure or 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, and
3. reduce the risks associated with the construction of the software.

#### Why Is Architecture Important?

Bass and his colleagues identify **three** key reasons that software architecture is important:

* + - * Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
      * The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
      * Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together” The architectural design model and the architectural patterns contained within it are transferable.

#### Architectural Descriptions

An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building.

The IEEE Computer Society has proposed, *Recommended Practice for Architectural Description of Software-Intensive Systems,* with the following objectives:

1. to establish a conceptual framework and vocabulary for use during the design of software architecture,
2. to provide detailed guidelines for representing an architectural description, and
3. to encourage sound architectural design practices.

The IEEE standard defines an *architectural description* (AD) as “a collection of products to document an architecture.” The description itself is represented using multiple views, where each *view* is “a representation of a whole system from the perspective of a related set of concerns.”

#### Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

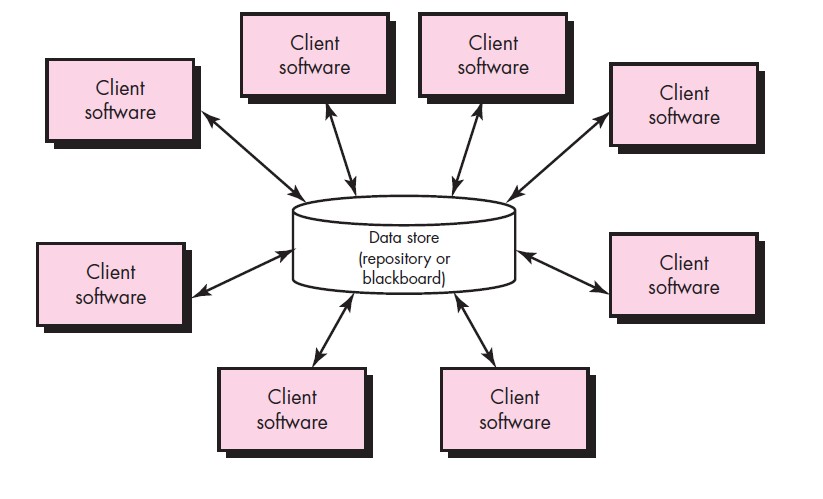
### ARCHITECTURAL STYLES

An *architectural style* as a descriptive mechanism to differentiate the house from other styles. The software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses (1) a set of components (e.g., a database, computational modules) that perform a function required by a system; (2) a set of connectors that enable “communication, coordination and cooperation” among components; (3) constraints that define how components can be integrated to form the system; and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

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.

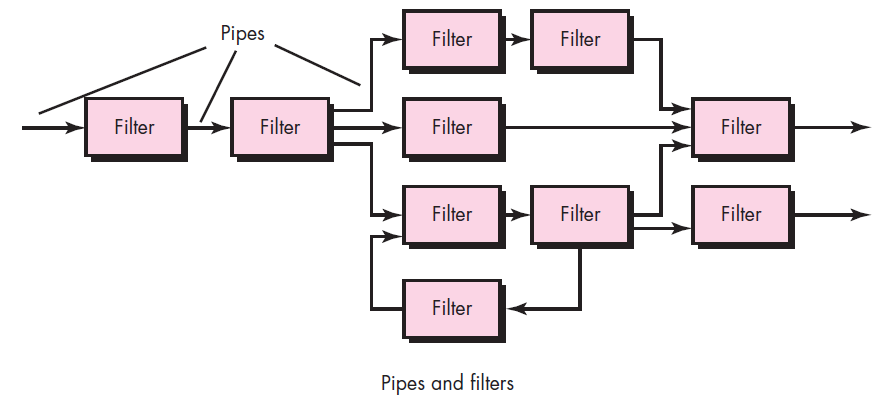
#### A Brief Taxonomy of Architectural Styles

**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. The following figure illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. Data-centered architectures promote ***integrability.***



#### Fig : Data-centered architecture

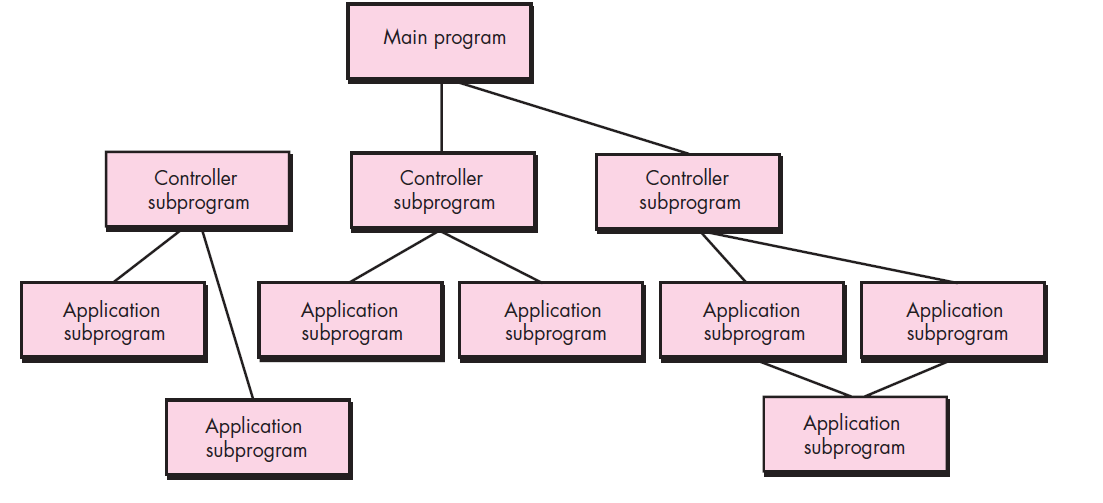
**Data-flow architectures.** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern shown in following figure. It has a set of components, called ***filters***, connected by ***pipes*** 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 of a specified form. However, the filter does not require knowledge of the Workings of its neighboring filters.



#### Fig : Data-flow architecture

**Call and return architectures.** This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category:

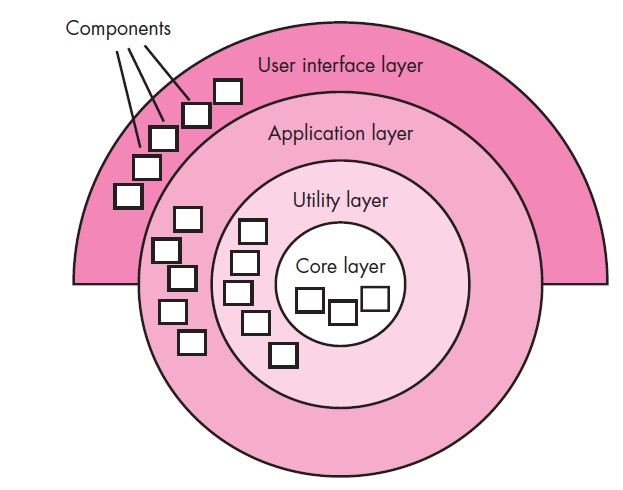
* + - * ***Main program/subprogram architectures****.* This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components that in turn may invoke still other components. The following figure illustrates an architecture of this type.
      * ***Remote procedure call architectures****.* The components of a main program/subprogram architecture are distributed across multiple computers on a network.



#### Fig : Main program/subprogram architecture

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

**Layered architectures**. The basic structure of a layered architecture is illustrated in following figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



#### Fig : Layered architecture

**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.

#### Organization and Refinement

The following questions provide insight into an architectural style:

**Control.** How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology? Is control synchronized or do components operate asynchronously?

**Data.** How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer? Do data components exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

### ARCHITECTURAL DESIGN

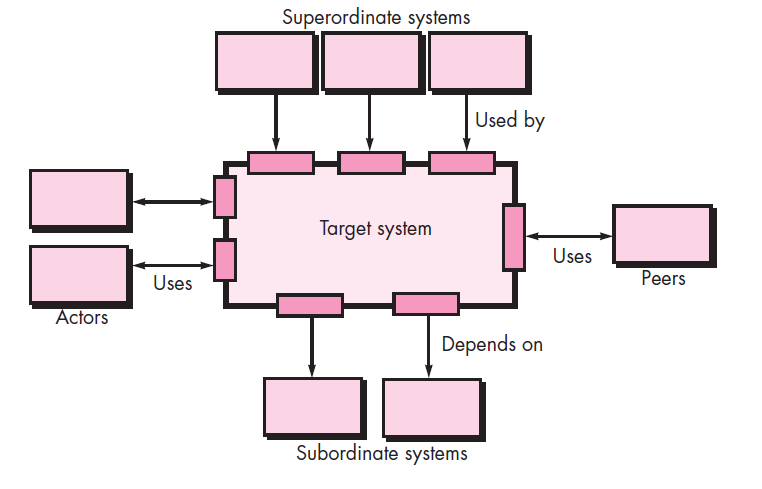
As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. Once context is modeled and all external software interfaces 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.

#### 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. The generic structure of the architectural context diagram is illustrated in following figure. Referring to the figure, 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 target system.
      * ***Actors***—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.



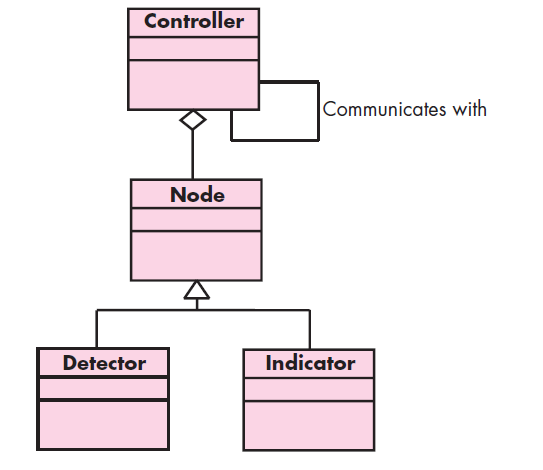
#### Fig : Architectural context diagram

**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.

The following archetypes can be used :

* + - * **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.
      * **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.
      * **Indicator.** An abstraction that represents all mechanisms (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 have the ability to communicate with one another.



#### Refining the Architecture into Components

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

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall architecture.

### ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

#### An Architecture Trade-Off Analysis Method

The Software Engineering Institute (SEI) has developed an ***architecture trade-off analysis method* (ATAM)** that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. ***Collect scenarios****.* A set of use cases is developed to represent the system from the user’s point of view.
2. ***Elicit requirements, constraints, and environment description****.* This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. ***Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.*** The architectural style(s) should be described using one of the following architectural views:
   * ***Module view*** for analysis of work assignments with components and the degree to which information hiding has been achieved.
   * ***Process view*** for analysis of system performance.
   * ***Data flow view*** for analysis of the degree to which the architecture meets functional requirements.
4. ***Evaluate quality attributes by considering each attribute in isolation****.* The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. ***Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.*** This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*
6. ***Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5****.*

#### Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system. Zhao suggests **three** types of dependencies:

***Sharing dependencies*** represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers

***Flow dependencies*** represent dependence relationships between producers and consumers of resources.

***Constrained dependencies*** represent constraints on the relative flow of control among a set of activities.

#### Architectural Description Languages

*Architectural description language* (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components.

### ARCHITECTURAL MAPPING USING DATA FLOW

. A mapping technique, called *structured design* is often characterized as a data flow- oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a **six** step process:

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.

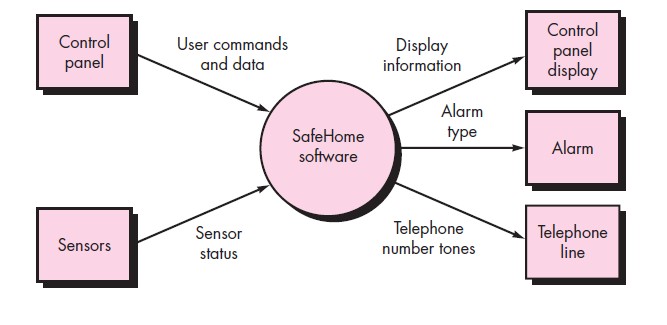
In order to perform the mapping, the type of information flow must be determined. One type of information flow is called ***transform flow*** and exhibits a linear quality. Data flows into the system along an ***incoming flow*** *path* where it is transformed from an external world representation into internalized form. Once it has been internalized, it is processed at a

***transform center****.* Finally, it flows out of the system along an ***outgoing flow path*** that transforms the data into external world.

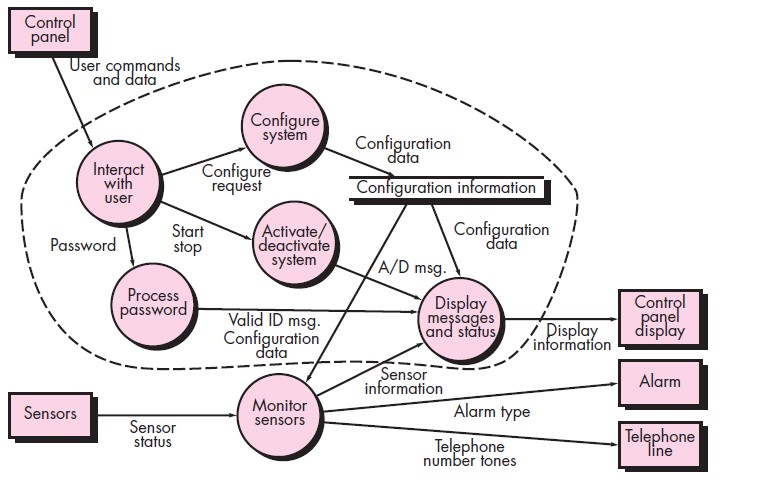
#### Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To map data flow diagrams into a software architecture, you would initiate the following design steps:

**Step 1**. **Review the fundamental system model. The fundamental system model :** The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. The following figure depicts a level 0 context model, and the next figure shows refined data flow for the security function.



#### Fig : Context-level DFD for the *SafeHome* security function



**Fig : Level 1 DFD for the *SafeHome* security function**

**Step 2. Review and refine data flow diagrams for the software.** Information obtained from the requirements model is refined to produce greater detail

**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 outgoing flow boundaries.** Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations.

**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.

**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 is 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.

**Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.** A 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 without difficulty, tested without confusion, and maintained without grief.

#### Refining the Architectural Design

Refinement of software architecture during early stages of design is to be encouraged. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

# Component-level design

Component-level design occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software.

### WHAT IS A COMPONENT?

A ***component*** is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* defines a component as “a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

The true meaning of the term ***component*** will differ depending on the point of view of the software engineer who uses it.

#### An Object-Oriented View

In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation. As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, you begin with the requirements model and elaborate analysis classes and infrastructure classes.

#### The Traditional View

In the context of traditional software engineering, a component is a functional element of a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a **module**, resides within the software architecture and serves one of **three** important roles:

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

### DESIGNING CLASS-BASED COMPONENTS

#### Basic Design Principles

**Four** basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied.

**The Open-Closed Principle (OCP).** *“****A module [component] should be open for extension but closed for modification****”* This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended without the need to make internal modifications to the component itself.

**The Liskov Substitution Principle (LSP).** *“****Subclasses should be substitutable for their base classes****”.* This design principle, originally proposed by Barbara Liskov, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it. In the context of this discussion, a “contract” is a ***precondition*** that must be true before the component uses a base class and a ***post condition*** that should be true after the component uses a base class.

**Dependency Inversion Principle (DIP).** *“****Depend on abstractions. Do not depend on concretions”***. The more a component depends on other concrete components, the more difficult it will be to extend.

**The Interface Segregation Principle (ISP).** *“****Many client-specific interfaces are better than one general purpose interface”***. ISP suggests that you should create a specialized interface to serve each major category of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

**The Release Reuse Equivalency Principle (REP).** *“****The granule of reuse is the granule of release”***. When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it. The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version. Rather than

addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.

**The Common Closure Principle (CCP).** *“****Classes that change together belong together.”*** Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

**The Common Reuse Principle (CRP).** *“****Classes that aren’t reused together should not be grouped together”***. When one or more classes within a package changes, the release number of the package changes. All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident. If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed.

#### Component-Level Design Guidelines

Ambler suggests the following guidelines:

**Components.** Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.

**Interfaces.** Interfaces provide important information about communication and collaboration. 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.

#### Cohesion

cohesion is the “single-mindedness” of a component. Lethbridge and Laganiére define a number of different types of cohesion

**Functional.** Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.

**Layer.** Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

**Communicational.** All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

#### Coupling

***Coupling*** is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep **coupling as low as is possible**.

Class coupling can manifest itself in a variety of ways. Lethbridge and Laganiére define the following coupling categories:

**Content coupling.** Occurs when one component “surreptitiously modifies data that is internal to another component”.

**Common coupling.** Occurs when a number of components all make use of a global variable. Although this is sometimes necessary, common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

**Control coupling.** Occurs when operation *A()* invokes operation *B()* and passes a control flag to

*B.* The control flag then “directs” logical flow within *B.* The problem with this form of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.

**Stamp coupling.** Occurs when **ClassB** is declared as a type for an argument of an operation of **ClassA**. Because **ClassB** is now a part of the definition of **ClassA**, modifying the system becomes more complex.

**Data coupling.** Occurs when operations pass long strings of data arguments. The “bandwidth” of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

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

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

**Inclusion or import coupling.** Occurs when component **A** imports or includes a package or the content of component **B**.

**External coupling.** Occurs when a component communicates or collaborates with infrastructure components. Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to **reduce coupling whenever possible**.

### CONDUCTING COMPONENT-LEVEL DESIGN

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

**Step 1. Identify all design classes that correspond to the problem domain.** Using the requirements and architectural model, each analysis class and architectural component is elaborated.

**Step 2. Identify all design classes that correspond to the infrastructure domain.** These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point.

**Step 3. Elaborate all design classes that are not acquired as reusable components.** Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

**Step 3a. Specify message details when classes or components collaborate.** The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that show how components within the system communicate and collaborate.

**Step 3c. Elaborate attributes and define data types and data structures required to implement them.** In general, data structures and types used to define attributes are defined within the context of the programming language that is to be

**Step 3d. Describe processing flow within each operation in detail.** This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each

software component is elaborated through a number of iterations that apply the stepwise refinement concept.

**Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.** Databases and files normally transcend the design description of an individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

**Step 5. Develop and elaborate behavioral representations for a class or component.** UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class.

**Step 6. Elaborate deployment diagrams to provide additional implementation detail.** Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions (often represented as subsystems) are represented within the context of the computing environment that will house them. During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components.

**Step 7. Refactor every component-level design representation and always consider alternatives.** The first component-level model you create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model.

### COMPONENT-LEVEL DESIGN FOR WEBAPPS

A WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

#### Content Design at the Component Level

Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user.

#### Functional Design at the Component Level

Modern Web applications deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion,

(2) provide computation or data processing capability that is appropriate for the WebApp’s business domain, (3) provide sophisticated database query and access, or (4) establish data interfaces with external corporate systems. To achieve capabilities, you will design and construct WebApp functional components that are similar in form to software components for conventional software.

WebApp functionality is delivered as a series of components developed in parallel with the information architecture to ensure that they are consistent.

During architectural design, WebApp content and functionality are combined to create a functional architecture. A ***functional architecture*** is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.

### DESIGNING TRADITIONAL COMPONENTS

The foundations of component-level design for traditional software components were formed in the early 1960s and were solidified with the work of Edsger Dijkstra and his colleagues. In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized “maintenance of functional domain.”

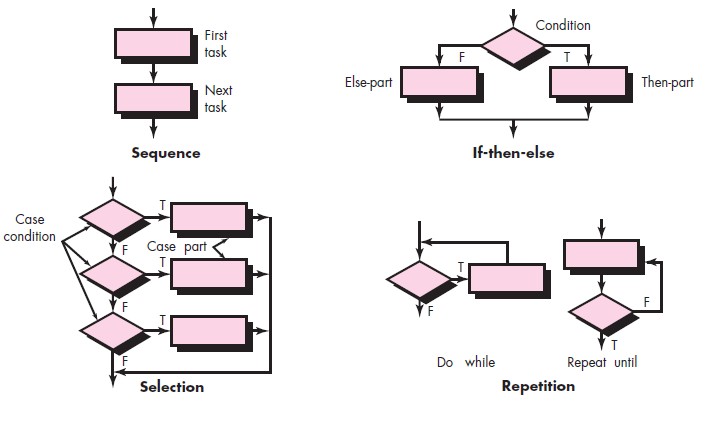
The constructs are **sequence, condition, and repetition**. ***Sequence*** implements processing steps that are essential in the specification of any algorithm. ***Condition*** provides the facility for selected processing based on some logical occurrence, and ***repetition*** allows for looping. These **three** constructs are fundamental to ***structured programming***—an important component-level design technique.

#### Graphical Design Notation

”A picture is worth a thousand words,” but it’s rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail.

The activity diagram allows you to represent **sequence, condition, and repetition** and all elements of **structured programming. A**nd is a descendent of an earlier pictorial design

representation called a ***flowchart.*** 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 . The following figure illustrates **three** structured constructs.



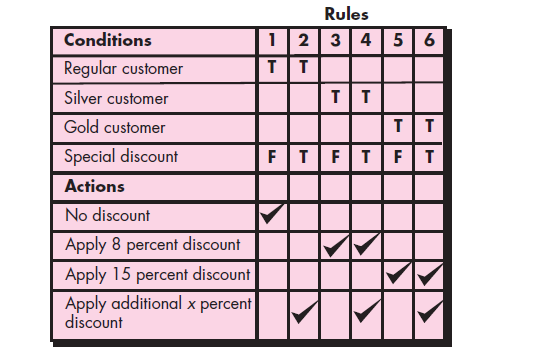
#### Fig : Flowchart constructs

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****.*

#### Tabular Design Notation

***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. Decision table organization is illustrated in following figure.. Referring to the figure, 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. Therefore, each column of the matrix may be interpreted as a ***processing rule****.* The following steps are applied to develop a decision table:

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

#### Fig : Decision table

**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, inter process synchronization, and many other features.

### COMPONENT-BASED DEVELOPMENT

***Component-based software engineering* (CBSE)** is a process that emphasizes the design and construction of computer-based systems using reusable software“ components.”

#### Domain Engineering

The intent of ***domain engineering*** is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components to reuse them during work on new and existing systems. Domain engineering includes **three** major activities— **analysis, construction, and dissemination**. The overall approach to ***domain analysis*** is often characterized within the context of object-oriented software engineering. The steps in the process are defined as:

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample and define analysis classes.
5. Develop a requirements model for the classes.

#### Component Qualification, Adaptation, and Composition

Domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in- house, others can be extracted from existing applications, and still others may be acquired from third parties.

**Component Qualification.** Component qualification ensures that a candidate component will

perform the function required, will properly “fit” into the architectural style specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application.

Among the many factors considered during component qualification are :

#### Application programming interface (API).

* + - * **Development and integration tools** required by the component.
      * **Run-time requirements**, including resource usage (e.g., memory or storage), timing or speed, and network protocol.
      * **Service requirements**, including operating system interfaces and support from other components.
      * **Security features**, including access controls and authentication protocol. • Embedded design assumptions, including the use of specific numerical or non numerical algorithms.
      * **Exception handling**. Each of these factors is relatively easy to assess when reusable components that have been developed in-house are proposed.

**Component Adaptation : A**n adaptation technique called ***component wrapping.*** When a software team has full access to the internal design and code for a component ***white-box wrapping*** is applied. **white-box** wrapping examines the internal processing details of the component and makes code-level modifications to remove any conflict. ***Gray-box wrapping*** is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. ***Black-box wrapping*** requires the introduction of pre- and post processing at the component interface to remove or mask conflicts.

**Component Composition.** The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure must be established to bind the components into an operational system. The infrastructure provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

#### Analysis and Design for Reuse

Binder suggests a number of key issues that form a basis for design for reuse:

**Standard data.** The application domain should be investigated and standard global data structures (e.g., file structures or a complete database) should be identified. All design components can then be characterized to make use of these standard data structures.

**Standard interface protocols.** Three levels of interface protocol should be established: the nature of intra modular interfaces, the design of external technical (nonhuman) interfaces, and the human-computer interface.

**Program templates.** An architectural style is chosen and can serve as a template for the architectural design of a new software.

#### Classifying and Retrieving Components

A reusable software component can be described in many ways, but an ideal description encompasses the ***3C model***—**concept, content, and context.**

The ***concept*** of a software component is “a description of what the component does”. The interface to the component is fully described and the semantics represented within the context of pre- and post conditions is identified. The concept should communicate the intent of the component.

The ***content*** of a component describes how the concept is realized. In essence, the content is information that is hidden from casual users and need be known only to those who intend to modify or test the component.

The ***context*** places a reusable software component within its domain of applicability. That is, by specifying conceptual, operational, and implementation features, the context enables a software engineer to find the appropriate component to meet application requirements.

A reuse environment exhibits the following characteristics:

* A component database capable of storing software components and the classification information necessary to retrieve them.
  + A library management system that provides access to the database.
  + A software component retrieval system that enables a client application to retrieve components and services from the server.
  + CBSE tools that support the integration of reused components into a new design or implementati

**UNIT- V**

**Software Testing strategies**: A strategic approach to software testing, Test strategies for conventional software, Validation testing, System testing, The Art of debugging.

**Product metrics:** A Framework for Product metrics, Metrics for the Requirements Model, Metrics for the Design Model, Metrics for Source code, Metrics for Testing, Metrics for Maintenance

# SOFTWARE TESTING STRATEGIES

### A STRATEGIC APPROACH TO SOFTWARE TESTING

A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:

* + - To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
    - Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
    - Different testing techniques are appropriate for different software engineering approaches and at different points in time.
    - Testing is conducted by the developer of the software and (for large projects) an independent test group.
    - Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

#### Verification and Validation

Software testing is one element of a broader topic that is often referred to as **verification** and **validation** (V&V). ***Verification*** refers to the set of tasks that ensure that software correctly implements a specific function. ***Validation*** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Boehm states this another way:

#### Verification: “Are we building the product right?” Validation: “Are we building the right product?”

Verification and validation includes a wide array of **SQA** activities: **technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing**.

#### Organizing for Software Testing

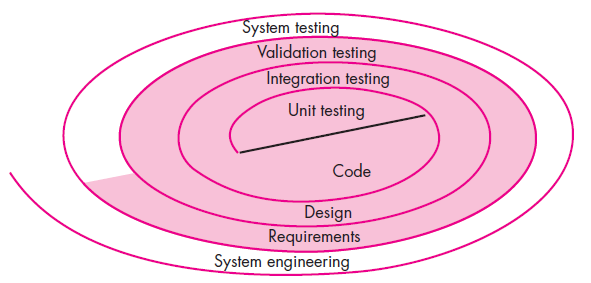
For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture. Only after the software architecture is complete does an independent test group become involved.

The role of an ***independent test group* (ITG)** is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

#### Software Testing Strategy—The Big Picture

The software process may be viewed as the spiral illustrated in following figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counter clockwise) along streamlines that decrease the level of abstraction on each turn.

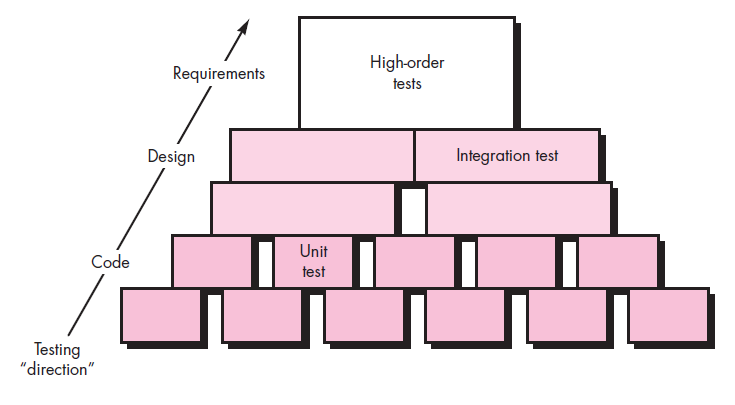


#### Fig : Testing Strategy

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

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of **four** steps that are implemented sequentially. The steps are shown in following figure. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing****.* Unit testing makes heavy use of testing techniques that exercise specific paths in a component’s control structure to ensure complete coverage and maximum error detection.

Next, components must be assembled or integrated to form the complete software package. ***Integration testing*** addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria must be evaluated. *Validation testing* provides final assurance that software meets all informational, functional, behavioral, and performance requirements.



#### Fig : Software testing steps

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). ***System testing*** verifies that all elements mesh properly and that overall system function/performance is achieved.

#### Criteria for Completion of Testing

“When are we done testing—how do we know that we’ve tested enough?” Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

One response to the question is: “You’re never done testing; the burden simply shifts from you (the software engineer) to the end user.” Every time the user executes a computer program, the program is being tested.

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The ***clean room software engineering*** approach suggests statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population.

. By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: “When are we done testing?”

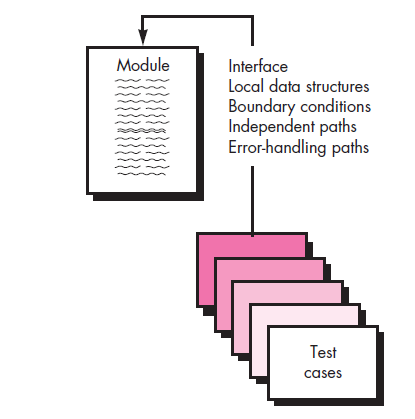
### TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

A testing strategy that is chosen by most software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

#### Unit Testing

***Unit testing*** focuses verification effort on the smallest unit of software design. 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.

**Unit-test considerations.** Unit tests are illustrated schematically in following figure. The module **interface** is tested to ensure that information properly flows into and out of the program unit under test. **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm’s execution. All **independent paths** through the control structure 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 operatesproperly at boundaries established to limit or restrict processing. And finally, all **error-handling paths** are tested.



#### Fig : Unit Test

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is invoked, when the maximum or minimum allowable value is encountered.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon calls this approach ***antibugging****.*

Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

**Unit-test procedures.** Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated.

The unit test environment is illustrated in following figure.. In most applications a ***driver*** is nothing more than a “main program” that accepts test case data, passes such data to the component (to be tested), and prints relevant results. ***Stubs*** serve to replace modules that are subordinate (invoked by) the component to be tested.

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.

#### Integration Testing

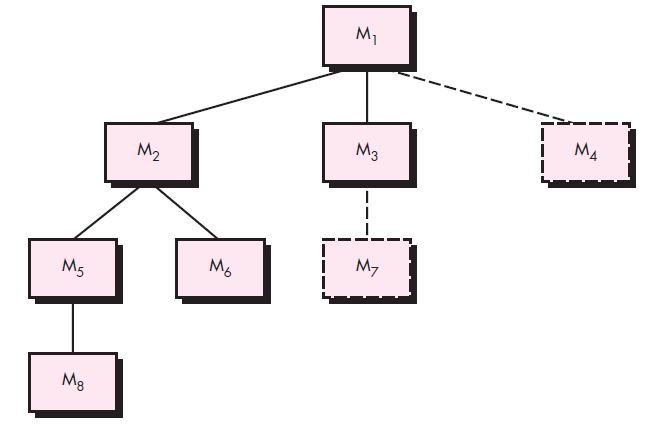
Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non incremental integration; that is, to construct the program using a “**big bang**” approach. All components are combined in advance. The entire program is tested as a **whole**. If 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.

Incremental integration is the antithesis of the big bang approach. 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. There are **two** different incremental integration strategies :

**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 to the main control module are incorporated into the structure in either a **depth-first or breadth-first** manner. Referring to the following figure, ***depth-first integration*** integrates all components on a major control path of the program structure. For example, selectingthe left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or 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.



#### Fig : Top-down integration

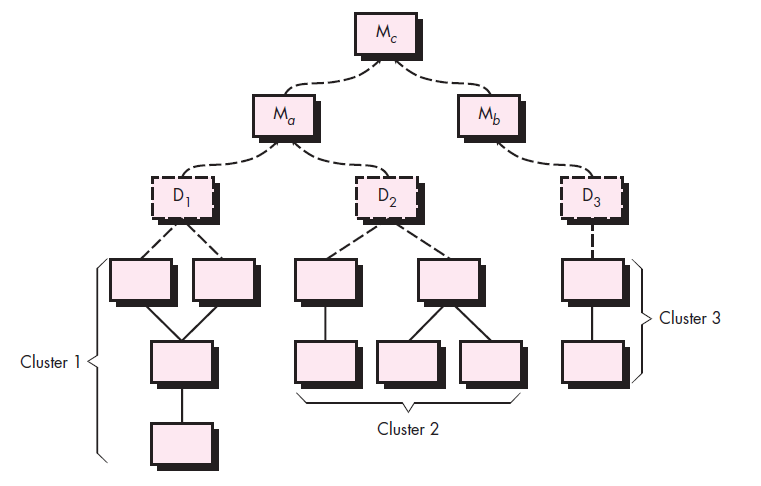
The integration process is performed in a series of **five** steps:

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

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

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

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



#### Fig : 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.

**Regression testing. R*egression testing*** is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression testing helps to ensure that changes 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:

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

As integration testing proceeds, the number of regression tests can grow quite large.

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

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

McConnell describes the smoke test in the following manner:

*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****.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
* ***The quality of the end product is improved****.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
* ***Error diagnosis and correction are simplified****.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with “new software increments”—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
* ***Progress is easier to assess****.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

### 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 can be defined in many ways, but a simple definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer.

#### Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created.

#### Configuration Review

An important element of the validation process is a ***configuration review****.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an **audit**

#### Alpha and Beta Testing

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

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.

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 that are encountered during beta testing and reports these to the developer at regular intervals.

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.

### SYSTEM TESTING

*System testing* is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

#### 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.

#### Security Testing

*Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

#### Stress Testing

Stress tests are designed to confront programs with abnormal situations. *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.

A variation of stress testing is a technique called ***sensitivity testing****.* Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

#### 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 tests are conducted. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

#### Deployment Testing

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

### THE ART OF DEBUGGING

*Debugging* occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art.

#### The Debugging Process

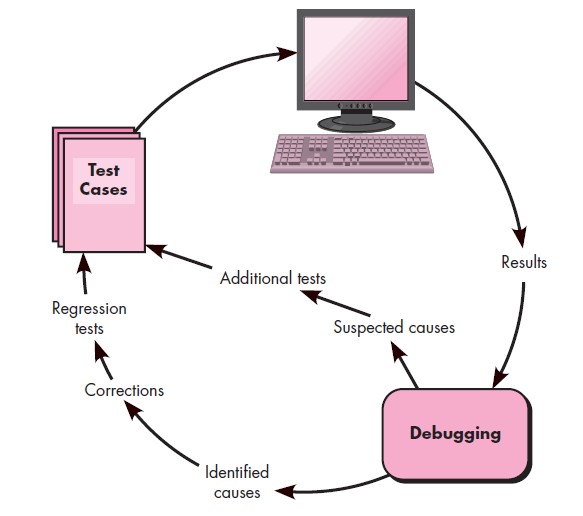
Debugging is not testing but often occurs as a consequence of testing. Referring to the following figure, the debugging process begins with the execution of a test case.. The debugging process attempts to match symptom with cause, thereby leading to error correction.

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.

A 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
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.



#### Fig : The Debugging Process

* + 1. **Psychological Considerations**

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren’t. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience.

#### Debugging Strategies

Bradley describes the debugging approach in this way:

Debugging is a straightforward application of the scientific method that has been developed over 2,500 years. The basis of debugging is to locate the problem’s source [the cause] by binary partitioning, through working hypotheses that predict new values to be examined. In general, **three** debugging strategies have been proposed

* + - 1. brute force,
      2. backtracking, and
      3. cause elimination.

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

#### Debugging tactics.

The ***brute force*** category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when **all else fails**.

***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.

The third approach to debugging is ***cause elimination.*** It is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence

#### 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. Van Vleck suggests **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****?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.
2. ***What “next bug” might be introduced by the fix I’m about to make?*** Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.
3. ***What could we have done to prevent this bug in the first place?*** This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

**Objective Questions UNIT- I**

1. Software delivers the most important product of our time is called **information**
2. **Data structures** Enable the programs to adequately manipulate information
3. Software is **developed or engineered**, it is not manufactured in the Classical Sense.
4. Software doesn’t **“Wear Out”**
5. **System software** is a collection of programs written to service other programs. Some system software
6. **Application software** is stand-alone programs that solve a specific business need. Application
7. **Product-line software** is designed to provide a specific capability for use by many different customers.
8. **Artificial intelligence** software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis.
9. ***Legacy*** software is older programs that are developed decades ago.
10. **Design** is a pivotal Software Engineering activity
11. The foundation for software engineering is the ***process* layer**.
12. A ***process*** is a collection of **activities, actions, and tasks** that are performed when some work product is to be created.
13. A process framework establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities***.
14. The process framework encompasses a set of ***umbrella activities*** that are applicable across the entire software process.
15. An ***iterative*** process flow repeats one or more of the activities before proceeding to the next.
16. An ***evolutionary*** process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software.
17. A ***process pattern*** describes a process-related problem that is encountered during software engineering work.
18. **Ambler** has proposed a template for describing a process pattern
19. ***Stage*** pattern defines a problem associated with a framework activity for the process.
20. ***Phase*** pattern define the sequence of framework activities that occurs within the process,
21. ***(SCAMPI*** provides a **five** step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning.
22. The ***waterfall*** model*,* sometimes called the classic life cycle,
23. A variation in the representation of the waterfall model is called the ***V-model*.**
24. **Incremental** development is particularly useful when staffing is unavailable.
25. Evolutionary models are **iterative**.
26. The Spiral Model originally proposed by **Barry Boehm**.
27. The concurrent development model, sometimes called **concurrent engineering**.
28. **Formal methods** enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.
29. Unified divides into **five** phases:
30. The **inception** *phas****e*** of the UP encompasses both customer communication and planning activities.

## UNIT- II

1. **Inception** establish a basic understanding of the problem, the people who want a solution.
2. The term ***specification*** means different things to different people.
3. The primary requirements validation mechanism is the **technical** review.
4. A ***stakeholder*** is anyone who has a direct interest in or benefits from the system that is to be developed.
5. Questions asked at the inception of the project should be “**context free**”
6. A **facilitator** controls the meeting.
7. *Quality function deployment* (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
8. ***Actors*** are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.
9. The system is described from the user’s point of view using a **scenario-based** approach.
10. **Analysis** patterns suggest solutions within the application domain that can be reused when modeling many applications.
11. The best negotiations strive for a “**win-win**” result.
12. **Flow-oriented** models that represent the functional elements of the system and how they transform data as it moves through the system
13. **Behavioral** models that depict how the software behaves as a consequence of external “events”
14. A UML **activity** diagram represents the actions and decisions that occur as some function is performed.
15. The most widely used data Model by the Software engineers is **Entity-Relationship**

*Diagram*

1. A **data object** is a representation of composite information that must be understood by software.
2. A data object encapsulates **data** only
3. **Operations** define the behavior of an object.
4. ***Class-responsibility-collaborator (CRC)*** modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.
5. A CRC model is really a collection of standard **index** cards that represent classes.
6. **Boundary** classes are used to create the interface that the user sees and interacts with as the software is used
7. **Controller** classes manage a “unit of work” from start to finish.
8. An **association** defines a relationship between classes.
9. **Multiplicity** defines how many of one class are related to how many of another class.
10. Dependencies are defined by a **stereotype**.
11. The ***data flow diagram* (DFD)** is the representation of Flow-oriented modeling.
12. The **process specification (PSPEC)** is used to describe all flow model processes that appear at the final level of refinement.
13. The **behavioral** model indicates how software will respond to external events
14. **Navigation** model defines the overall navigation strategy for the WebApp.
15. **Configuration** model—describes the environment and infrastructure in which the WebApp resides.

### UNIT- III

1. The **data/class** design transforms class models into design class realizations and the requisite data structures required to implement the software.
2. The **architectural** design defines the relationship between major structural elements of the software.
3. The **component-level** design transforms structural elements of the software architecture into a procedural description of software components.
4. The importance of software design can be stated with a single word **quality.**
5. **Performance** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
6. A **procedural** abstraction refers to a sequence of instructions that have a specific and limited function.
7. A **data** abstraction is a named collection of data that describes a data object.
8. **Structural** models represent architecture as an organized collection of program components.
9. **Functional** models can be used to represent the functional hierarchy of a system.
10. Software is divided into separately named and addressable components, sometimes called **module*.***
11. **Cohesion** is an indication of the relative functional strength of a module.
12. **Coupling** is an indication of the relative interdependence among modules.
13. Refinement is actually a process of **elaboration.**
14. An **aspect** is a representation of a crosscutting concern.
15. Data-centered architectures promote **integrability**
16. **Sharing** dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers
17. ***Architectural description language* (ADL)** provides a semantics and syntax for describing a software architecture.
18. A **component** is a modular building block for computer software.
19. A module should be open for extension but closed for **modification**
20. **External** coupling Occurs when a component communicates or collaborates with infrastructure components.

### UNIT- IV

1. Reduce demand on **short-term** memory**.**
2. A human engineer establishes a **user** mode
3. The user’s **mental** model is the image of the system that end users carry in their heads.
4. **Variability** refers to the deviation from average response time
5. Error messages and warnings are **bad news**
6. Don’t be afraid of **white space**
7. Design is the engineering activity that leads to a **high-quality** product.
8. The typical end user expects WebApps to be available **24/7/365**.
9. **Content** architecture focuses on the manner in which content objects are structured for presentation and navigation.
10. A WebApp **hierarchical** structure can be designed in a manner that enables flow of control horizontally across vertical branches of the structure.
11. The **MVC** architecture decouples the user interface from WebApp functionality and information content.
12. The **controller** manages access to the model and the view and coordinates the flow of data between them
13. OOHDM **conceptual** design creates a representation of the subsystems, classes, and relationships that define the application domain for the WebApp.
14. **Navigational** design identifies a set of “objects” that are derived from the classes defined in conceptual design.
15. **OOHDM** uses a predefined set of navigation classes such as nodes, links, anchors, and access structures.
16. A **context** includes a description of the local navigation structure, restriction imposed on the access of content objects, and methods required to effect access of content objects.
17. The **abstract interface design** action specifies the interface objects that the user sees as WebApp interaction occurs.
18. A formal model of interface objects, called an ***abstract data view* (ADV**
19. The ADV model defines a “**static layout**
20. The OOHDM **implementation** activity represents a design iteration that is specific to the environment in which the WebApp will operate.

### UNIT -V

1. Testing is conducted by the developer of the software and an **independent test** group.
2. **Verification** refers to the set of tasks that ensure that software correctly implements a specific function.
3. **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
4. **Unit** testing begins at the vortex of the spiral and concentrates on each unit of the software as implemented in source code.
5. State testing objectives **explicitly***.*
6. **Top-down** integration testing is an incremental approach to construction of the software architecture.
7. Bottom-up integration testing*,* as its name implies, begins construction and testing with

**atomic *modules***

1. **R*egression* testing** is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
2. **Use-based** testing begins the construction of the system by testing those classes that use very few *server* classes.
3. **Cluster** testing is one step in the integration testing of OO software.
4. The **alpha** test is conducted at the developer’s site by a representative group of end users.
5. The **beta test** is conducted at one or more end-user sites.
6. **Stress** testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
7. Deployment testing*,* sometimes called **configuration** testing
8. **Debugging** occurs as a consequence of successful testing.
9. **Backtracking** is a fairly common debugging approach that can be used successfully in small programs.
10. **White-box testing***,* sometimes called glass-box testing
11. **Condition testing** is a test-case design method that exercises the logical conditions contained in a program module.
12. Black-box testing, also called **behavioral** testing*.*
13. **Cluster** testing is one step in the integration testing of OO software.

## Important Questions (10 Marks)

**UNIT- I**

1. What is Software ? Discuss its characteristics and its Applications

#### ( Ref : 1.1 @ page no : 2)

1. What is Software Engineering? Discuss Software Engineering – A Layard Technology

#### ( Ref : 1.3 @ page no : 7)

1. What are different software myths. Explain **( Ref : 1.6 @ page no : 12)**
2. What is Process Assessment Method for Improvement for the software process.

#### ( Ref : 1.8 @ page no : 19)

1. Briefly discuss about Prescriptive process models **( Ref : 1.9 @ page no : 20)**
2. Explain PSP and TSP process models **( Ref : 1.12 @ page no : 31)**
3. Discuss Extreme programming (xp) **( Ref : 1.18 @ page no : 37)**
4. What is unified process ? Discuss different phases in the unified process

#### ( Ref : 1.11 @ page no : 29)

1. What is Agile Process ? List Agility Principles **( Ref : 1.17 @ page no : 35)**

## Unit-II

1. What is Requirements Engineering ? Explain different tasks involved in Requirement Engineering Process. **( Ref : 2.1 @ page no : 50)**
2. How do you develop Use Cases **( Ref : 2.4 @ page no : 56)**
3. List the elements of Requirements Model **( Ref : 2.5.1 @ page no : 57)**
4. Discuss Data Modeling Concepts **( Ref : 2.11 @ page no : 68)**
5. Explain Flow Oriented Modeling **( Ref : 2.14 @ page no : 76)**
6. Discuss Scenario-Based Modeling **( Ref : 2.9 @ page no : 63)**

## Unit-III

1. Explain Design within the Context of Software Engineering and discuss design process and design quality **( Ref : 3.1 @ page no : 87)**
2. Discuss Design concepts **( Ref : 3.3 @ page no : 91)**
3. What is design model ? discuss four design elements **( Ref : 3.4 @ page no : 96)**
4. Discuss Architectural Styles and Patterns **( Ref : 3.7 @ page no : 101)**
5. (a) List Basic Design Principles and guidelines for Component-level Design

(b) What is Coupling and how it is differ from Cohesion **( Ref : 3.12 @ page no : 114)**

## Unit-IV

1. State and explain Golden Rules of UI ? How these rules affect on UI analysis and design.

#### ( Ref : 4.1 & 4.2 @ page no : 125 & 127)

1. (a) Explain about Inter design evaluation. **( Ref : 4.6 @ page no : 139)**

(b) List Interface Design Goals **( Ref : 4.8 @ page no : 142)**

1. Discuss Webapp Architecture **( Ref : 4.13.2 @ page no : 150)**
2. What is Component level Design ? Explain OOHDM **( Ref : 4.15 & 4.16 @ page no : 153)**

## Unit-V

1. Discuss Software Testing Strategies for Conventional Software

#### ( Ref : 5.3 @ page no : 159)

1. Discuss The Art of Debugging **( Ref : 5.8 @ page no : 169)**
2. Discuss White Box Testing **( Ref : 5.11, 5.12 & 5.13 @ page no : 174-180)**
3. Discuss Black Box Testing **( Ref : 5.14 @ page no : 180)**

## Important Questions (2 Marks)

**UNIT- I**

1. Explain framework activities and umbrella activities for software process

#### ( Ref : 1.4 @ page no : 8)

1. What are the disadvantages of formal Methods model ( **Ref : 1.10.2 @ page no :27)**
2. What are the general principles that are focus on Software Engineering practice

#### ( Ref : 1.5.1 @ page no :11)

1. What is process pattern ( **Ref : 1.7.3 @ page no :17)**

## UNIT- II

1. What is Quality Function Development ( **Ref : 2.3.2 @ page no :55)**
2. What is CRC Modeling **( Ref : 2.12.4 @ page no : 71)**
3. What is Process Specification **( Ref : 2.14.4 @ page no : 80)**
4. Explain Associations and Dependencies **( Ref : 2.12.5 @ page no : 50)**

## UNIT- III

1. What are the software quality guidelines **( Ref : 3.2.1 @ page no : 89)**
2. What are Quality Attributes **( Ref : 3.2.1 @ page no : 90)**
3. What is architecture ? **( Ref : 3.5.1 @ page no : 99)**
4. Define Archetype **( Ref : 3.8.2 @ page no : 106)**
5. Define Component **( Ref : 3.11 @ page no : 113)**

## UNIT- IV

1. Design issues of UI **( Ref : 4.4.3 @ page no : 133).**
2. Quality Attributes for Webapp design **( Ref : 4.7 @ page no : 141)**
3. How do you conduct User Analysis **( Ref : 4.3.1 @ page no : 130)**
4. What is Abstract Interface Design and Implementation **( Ref : 4.16.3 @ page no : 154)**

## UNIT- V

1. Discuss testing strategy for small and large software testing **( Ref : 5.1.3 @ page no : 156)**
2. What is Verification & Validation **( Ref : 5.1.1 @ page no : 155)**
3. What is Alpha and Beta Testing **( Ref : 5.6.3 @ page no : 167)**
4. Explain System testing **( Ref : 5.7 @ page no : 168)**