LECTURE NOTES

ON

Software Engineering

II B.TECH II SEMESTER

(JNTUA-R15)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA

Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112

(Approved by AICTE, New Delhi Affiliated to JNTUA Ananthapuramu. ISO 9001:2015 Certified Institute)

UNIT- I

Software and Software Engineering: The Nature of Software, The Unique Nature of WebApps, Software Engineering, The Software Process, Software Engineering Practice, Software Myths
Process Models: A Generic Process Model, Process Assessment and Improvement, Prescriptive Process Models, Specialized Process Models, The Unified Process, Personal and Team Process Models, Process Technology, Product and Process.
Agile Development: Agility, Agility and the Cost of Change, Agile Process, Extreme Programming, Other Agile 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.

1.1 THE NATURE OF SOFTWARE

- Software takes Dual role of Software. It is a **Product** and at the same time a **Vehicle for delivering a product**.
- Software delivers the most important product of our time is called **information**

1.1.1 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.

Characteristics of software

Software has characteristics that are considerably different than those of hardware:

1) Software is developed or engineered, it is not manufactured in the Classical Sense.

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both the activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent or easily corrected for software. Both the activities are dependent on people, but the relationship between people is totally varying. These two activities require the construction of a "**product**" but the approaches are different. Software costs are concentrated in engineering which means that software projects cannot be managed as if they were manufacturing.

2) Software doesn't "Wear Out"

The following figure shows the relationship between failure rate and time. Consider the failure rate as a function of time for hardware. The relationship is called **the bathtub curve**, indicates that hardware exhibits relatively high failure rates early in its life, defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. So, stated simply, the hardware begins to wear out. Software is not susceptible to the environmental maladies that cause **hardware to wear out**

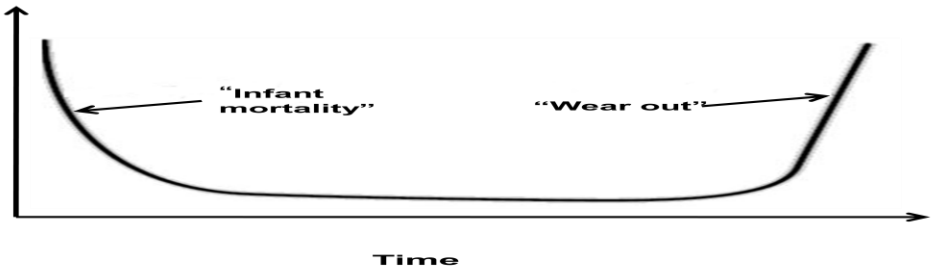


Fig: FAILURE CURVE FOR HARDWARE

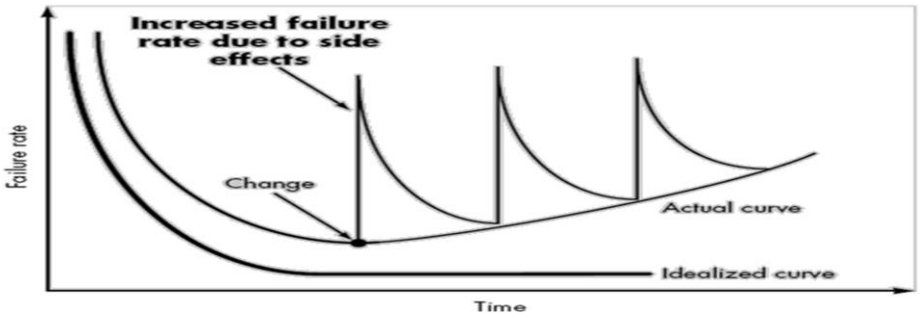


Fig: FAILURE CURVE FOR SOFTWARE

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

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts

1.1.2 Software Application Domains

Seven Broad Categories of software are challenges for software engineers:

(a) System software: A collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities)

(b)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).

(c)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.

(d) 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).

(e) 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).

(f) 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.

(g) 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.

New Software Challenges

- **Open-world computing** : Creating software to allow machines of all sizes to communicate with each other across vast networks (Distributed computing—wireless networks)
- **Net sourcing** : Architecting simple and sophisticated applications that benefit targeted end-user markets worldwide (the Web as a computing engine)
- **Open Source** : Distributing source code for computing applications so customers can make local modifications easily and reliably ("free" source code open to the computing community)

1.1.3 Legacy Software

- Legacy software is older programs that are developed decades ago.
- The quality of legacy software is poor because it has inextensible design, convoluted code, poor and nonexistent documentation, test cases and results that are not achieved.

As time passes legacy systems evolve due to following reasons:

- The software must be adapted to meet the needs of new computing environment or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with more modern systems or database
- The software must be re-architected to make it viable within a network environment.

****

1.2 UNIQUE NATURE OF WEB APPS

In the early days of the World Wide Web, websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics. As time passed, the augmentation of HTML by development tools (e.g., XML, Java) enabled Web engineers to provide computing capability along with informational content. *Web-based systems and applications* (*WebApps*) were born. Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

WebApps are one of a number of distinct software categories. Web-based systems and applications "involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."

The following attributes are encountered in the vast majority of WebApps.

- **Network intensiveness.** A WebApp resides on a network and must serve the needs of a diverse community of clients. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).
- **Concurrency.** A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.
- **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One hundred users may show up on Monday; 10,000 may use the system on Thursday.
- **Performance.** If a WebApp user must wait too long, he or she may decide to go elsewhere.
- **Availability.** Although expectation of 100 percent availability is un reasonable, users of popular WebApps often demand access on a 24/7/365 basis

- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.
- **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure modes
- **Aesthetics**. An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

****

1.3 SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities

- **Problem should be understood before software solution is developed**
- **Design is a pivotal Software Engineering activity**
- **Software should exhibit high quality**
- **Software should be maintainable**

These simple realities lead to one conclusion. Software in all of its forms and across all of its application domains should be **engineered**.

Software Engineering by Fritz Bauer defined as:
Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

IEEE has developed a more comprehensive definition as:

1) ***Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.***
2) ***The study approaches as in (1)***

Software Engineering is a **layered technology**. Software Engineering encompasses a **Process, Methods** for managing and engineering software and **tools**.

The following Figure represents **Software engineering Layers**

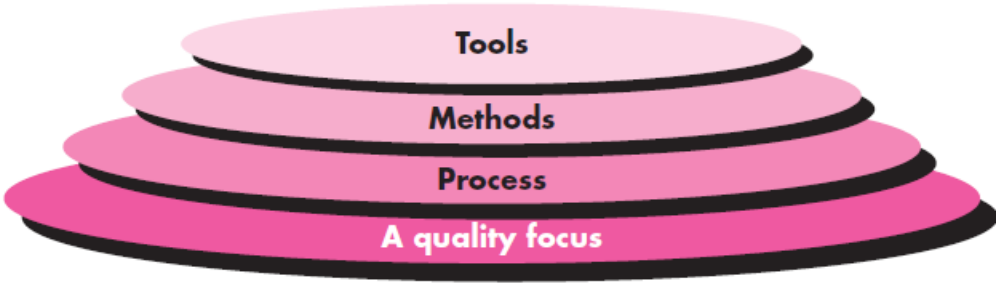


Fig: Software Engineering-A layered technology

Software engineering is a layered technology. Referring to above Figure, any engineering approach must rest on an organizational commitment to **quality**.

The bedrock that supports software engineering is a **quality focus**.

The foundation for software engineering is the ***process* layer**. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. **Process** defines a **framework** that must be established for effective delivery of software engineering technology.

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

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

****

1.4 THE SOFTWARE PROCESS

A ***process*** is a collection of **activities, actions, and tasks** that are performed when some work product is to be created.
An ***activity*** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.
An ***action*** encompasses a set of tasks that produce a major work product (e.g., an architectural design model).
A ***task*** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.
A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities*** that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of ***umbrella activities*** that are applicable across the entire software process.

A generic process framework for software engineering encompasses **five** activities:

- **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer. The intent is to understand stakeholders objectives for the project and to gather requirements that help define software features and functions.
- **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
- **Modeling.** Creation of models to help developers and customers understand the requires and software design
- **Construction.** This activity combines code generation and the testing that is required to uncover errors in the code.
- **Deployment.** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

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

Software engineering process framework activities are complemented by a number of U**mbrella Activities**. In general, **umbrella activities** are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

- **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
- **Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.
- **Software quality assurance**—defines and conducts the activities required to ensure software quality.
- **Technical reviews**—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
- **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders needs; can be used in conjunction with all other framework and umbrella activities.
- **Software configuration management**—manages the effects of change throughout the software process.

- **Reusability management**—defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
- **Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

Attributes for Comparing Process Models

- Overall flow and level of interdependencies among tasks
- Degree to which work tasks are defined within each framework activity
- Degree to which work products are identified and required
- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor of process description
- Degree to which stakeholders are involved in the project
- Level of autonomy given to project team
- Degree to which team organization and roles are prescribed

****

1.5 Software Engineering Practice

A generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—communication, planning, modeling, construction, and deployment—and umbrella activities establish a skeleton architecture for software engineering work.

1.5.1 The Essence of Practice

The essence of software engineering practice:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

Understand the problem. It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think, *Oh yeah, I understand, let's get on with solving this thing.* Unfortunately, understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

• *Who has a stake in the solution to the problem?* That is, who are the stakeholders?

• *What are the unknowns?* What data, functions, and features are required to properly solve the problem?

• *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?

• *Can the problem be represented graphically?* Can an analysis model be created?

Plan the solution. Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

• *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the
data, functions, and features that are required?

• *Has a similar problem been solved?* If so, are elements of the solution reusable?

• *Can sub problems be defined?* If so, are solutions readily apparent for the sub problems?

• *Can you represent a solution in a manner that leads to effective implementation?*
Can a design model be created?

Carry out the plan. The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

• *Does the solution conform to the plan?* Is source code traceable to the design model?

• *Is each component part of the solution provably correct?* Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

Examine the result. You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

• *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?

• *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

1.5.2 Software General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought."

David Hooker has proposed **seven** principles that focus on software Engineering practice.

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*.

The Second Principle: KISS (Keep It Simple, Stupid!)

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*.

The Third Principle: Maintain the Vision

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself.

The Fourth Principle: What You Produce, Others Will Consume

Always specify, design, and implement knowing someone else will have to understand what you are doing.

The Fifth Principle: Be Open to the Future

A system with a long lifetime has more value. *Never design yourself into a corner*. *Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.*

The Sixth Principle: Plan Ahead for Reuse

Reuse saves time and effort. *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

The Seventh principle: Think!

Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right.

****

1.6 SOFTWARE MYTHS

Software Myths- beliefs about software and the process used to build it - can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious. For instance, myths appear to be reasonable statements of fact, they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score"

Management Myths:

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth.

Myth1: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality:

- The book of standards may very well exist, but is it used?
- Are software practitioners aware of its existence?
- Does it reflect modern software engineering practice?
- Is it complete?
- Is it adaptable?
- Is it streamlined to improve time to delivery while still maintaining a focus on Quality?

In many cases, the answer to these entire question is NO.

Myth2: *If we get behind schedule, we can add more programmers and catch up*

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

Myth3: *If we decide to outsource the software project to a third party, I can just relax and let that firm build it.*
Reality: If an organization does not understand how to manage and control software project internally, it will invariably struggle when it out sources software project.

Customer Myths:

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing /sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths led to false expectations and ultimately, dissatisfaction with the developers.

Myth1: *A general statement of objectives is sufficient to begin writing programs - we can fill in details later.*
Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is a recipe for disaster. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.

Myth2: *Project requirements continually change, but change can be easily accommodated because software is flexible.*
Reality: It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early, cost impact is relatively small. However, as time passes, cost impact grows rapidly – resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's Myths:

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

Myth1: *Once we write the program and get it to work, our job is done.*
Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

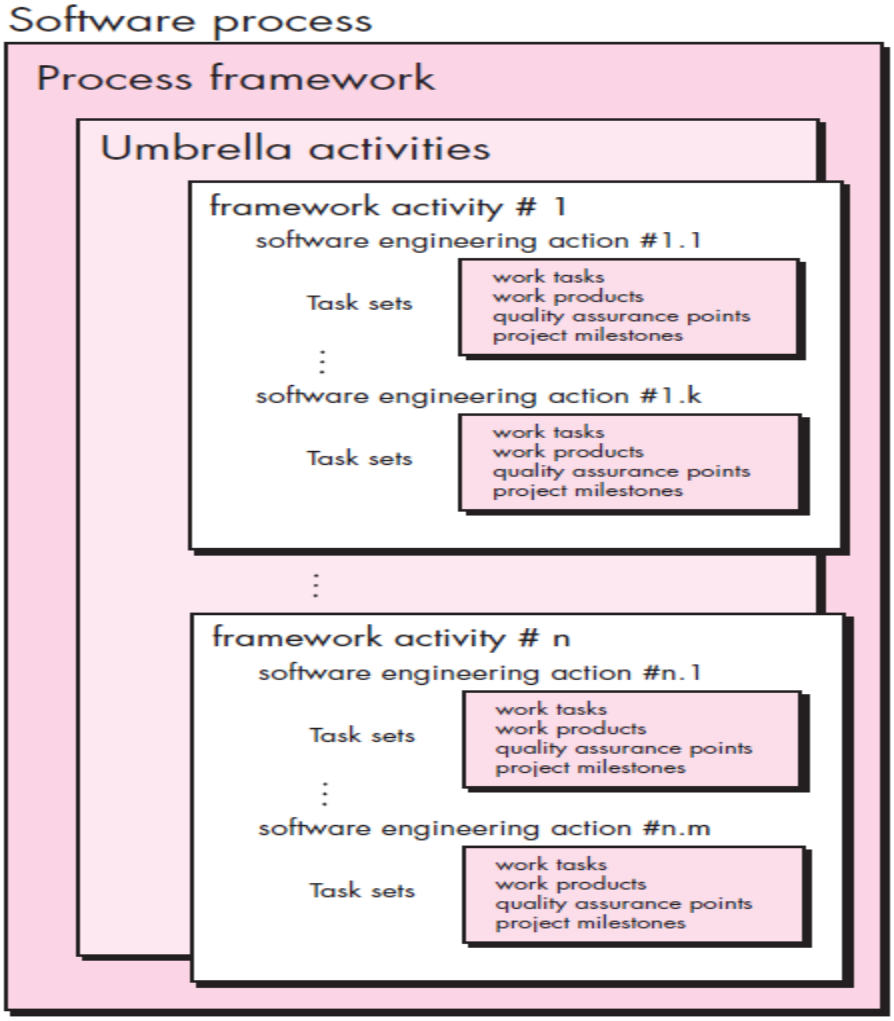
Myth2: *Until I get the program "running" I have no way of assessing its quality.*
Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the *formal technical review.* Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth3: *The only deliverable work product for a successful project is the working program.*
Reality: A working program is only one part of a *software configuration* that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

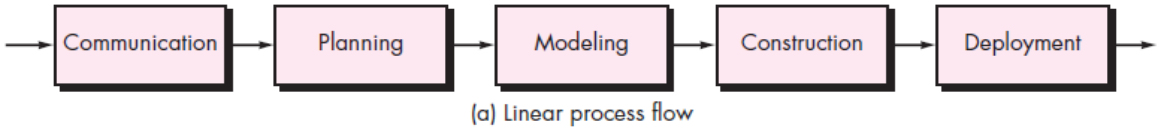
Myth4: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*
Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.7 A GENERIC PROCESS MODEL

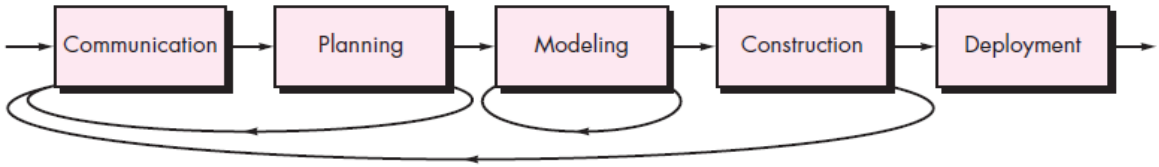
The software process is represented schematically in following figure. Each framework activity is populated by a set of software engineering actions. Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced,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



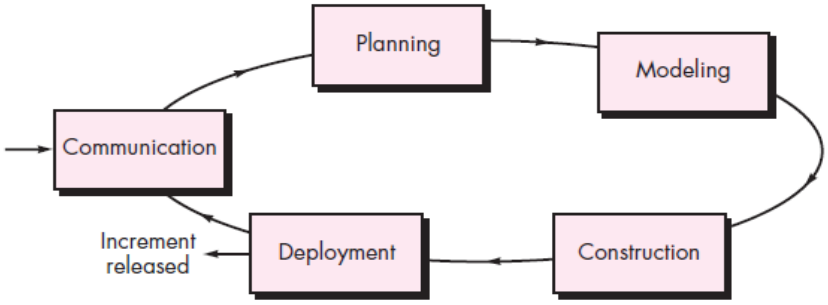
- A generic process framework defines **five** framework activities—**communication, planning, modeling, construction,** and **deployment.**
- In addition, a set of umbrella activities **project tracking and control, risk management, quality assurance, configuration management, technical reviews, and 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.
- 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).



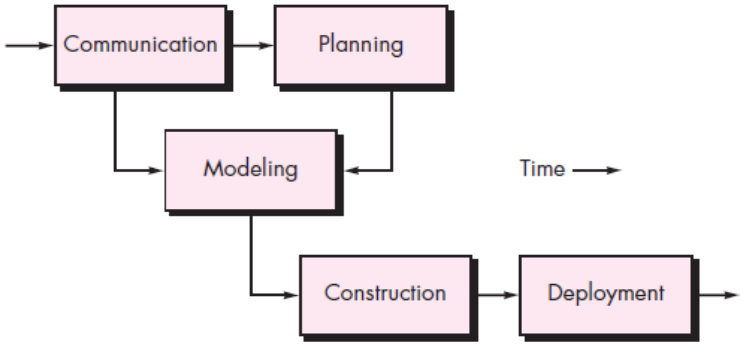
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

1.7.1 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:

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

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

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

1. ***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.

2. ***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).
3. ***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 **Spiral Model** or **Prototyping.**

(d) 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?

(e) Problem. The specific problem to be solved by the pattern.

(f) 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.

(g) 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?

(h) 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.

(i) 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).

****

1.8 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.

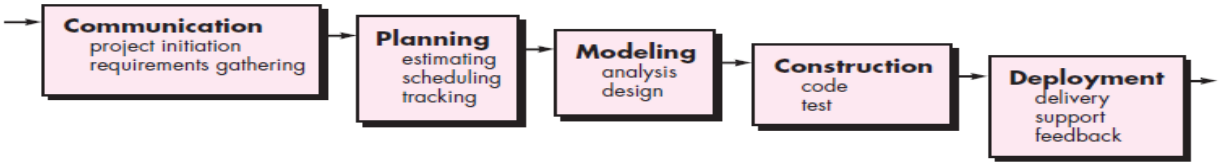
****

1.9 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.

1.9.1 The Waterfall Model

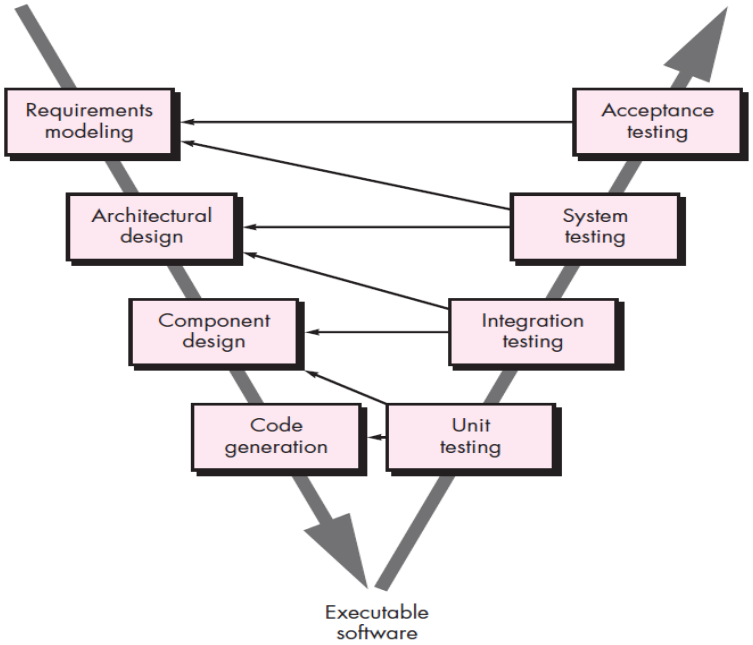
The ***waterfall model***, sometimes called the ***classic life cycle***, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through **planning, modeling, construction, and deployment**.



The waterfall model

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

The V-model



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:

1. 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.
2. 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.
3. 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.

1.9.2 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.

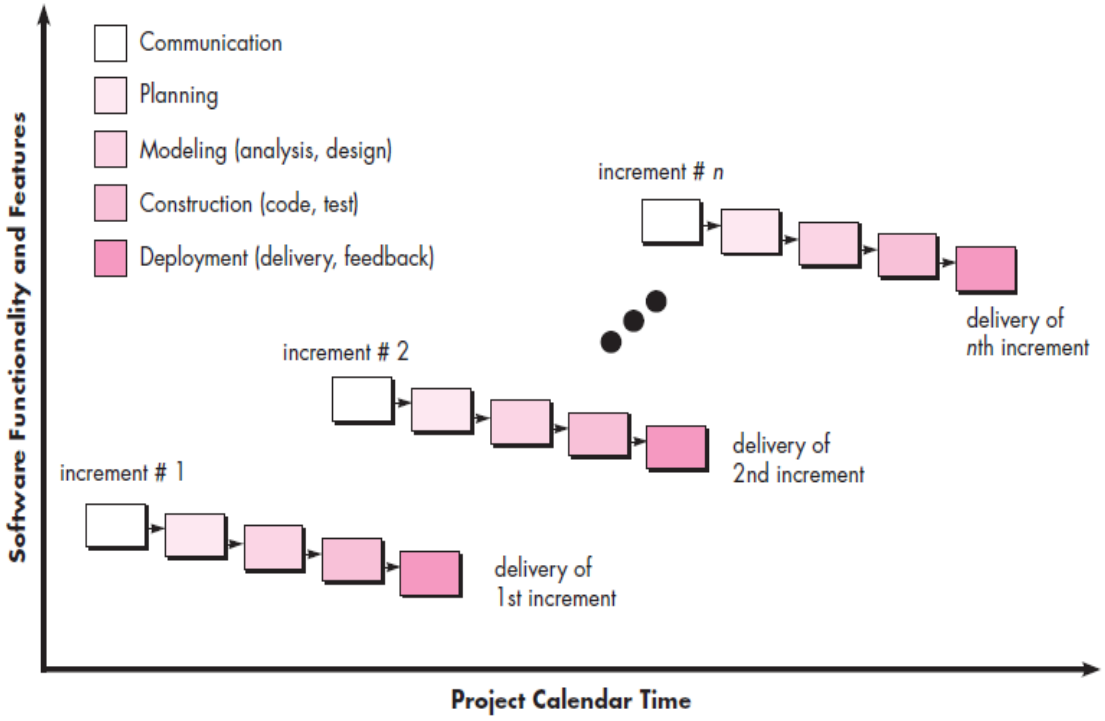


Fig : Incremental Model

1.9.3 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.

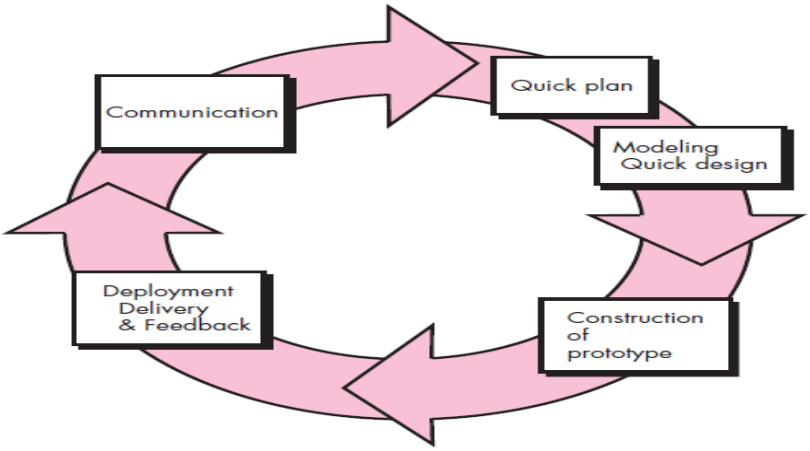


Fig : prototyping paradigm

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:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.
2. 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.

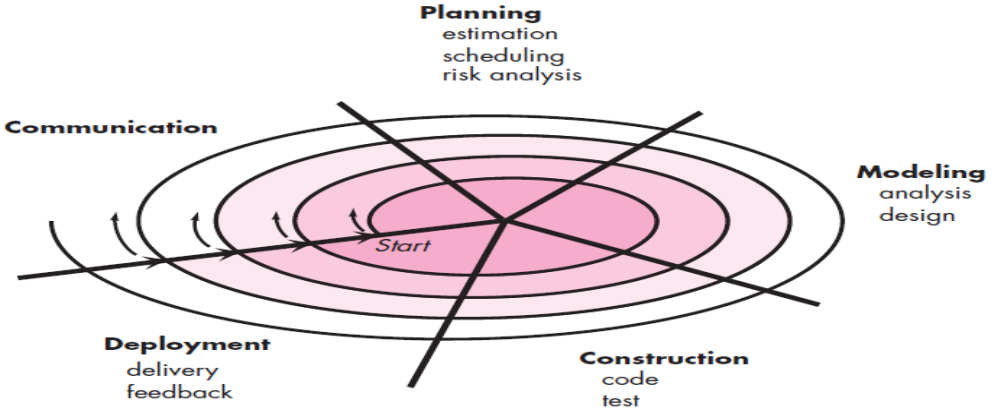


Fig : The Spiral Model

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.

1.9.4 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 provide 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.

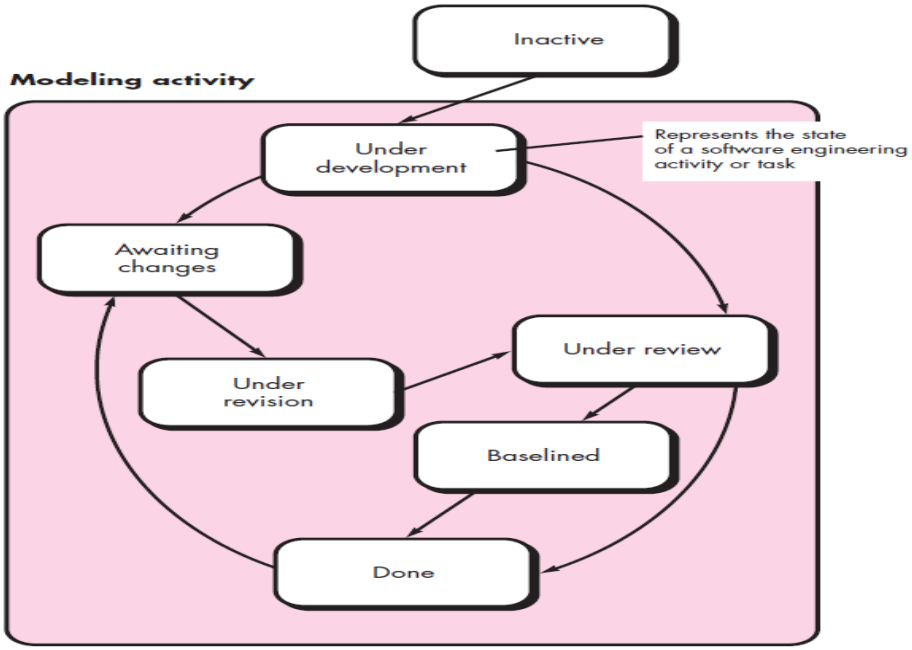


Fig : Concurrent development model

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.

****

1.10 SPECIALIZED PROCESS MODELS

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

1.10.1 Component-Based Development

The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from **prepackaged** software components.

Modeling and construction activities begin with the identification of **candidate components.** These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

1.10.2 The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called ***clean room software engineering***.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. **Ambiguity, incompleteness, and inconsistency** can be discovered and corrected more easily, but through the application of mathematical analysis.

When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected. Although not a mainstream approach, the formal methods model offers the promise
of **defect-free software**.

Draw Backs:

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

1.10.3 Aspect-Oriented Software Development

A OSD defines "aspects" that express customer concerns that cut across multiple system functions, features, and information. When concerns cut across multiple system functions, features, and information, they are often referred to as ***crosscutting concerns***. ***Aspectual requirements*** define those crosscutting concerns that have an impact across the software architecture.

Aspect-oriented software development (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for **defining, specifying, designing, and constructing *aspects***."

Grundy provides further discussion of aspects in the context of what he calls *aspect-oriented component engineering* (AOCE):

AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "**aspects**," to characterize cross-cutting functional and non-functional properties of components.

****

1.11 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.

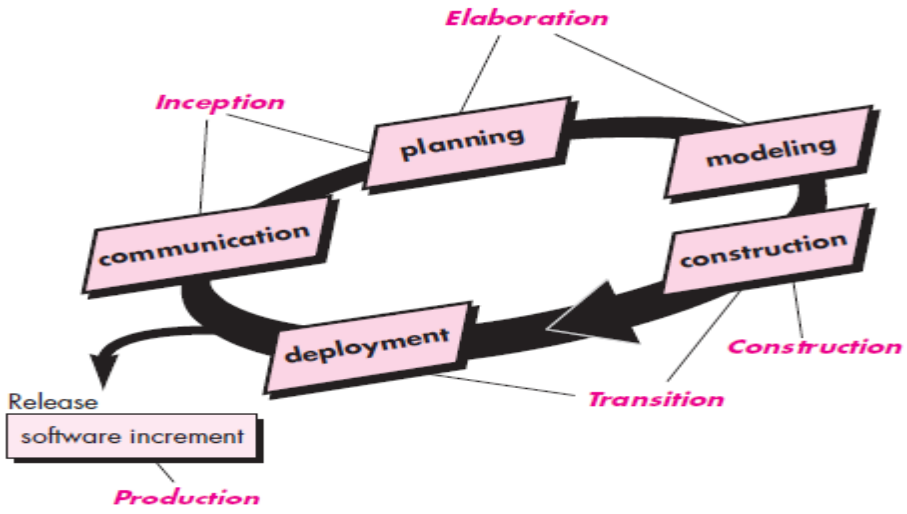
1.11.1 A Brief History

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**.

1.11.2 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.

****

1.12 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.

1.12.1 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.

1.12.2 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.

****

1.13 PROCESS TECHNOLOGY

Process technology tools allow a software organization to build an automated model of the process framework, task sets, and umbrella activities. The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.

Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model. Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

****

1.14 PRODUCT AND PROCESS

The **Product** is ***what*** **we're** actually building. What's our solution to the problem at hand? Half of engineering is making sure you're building the right product and have the ability to actually build it. For software engineers, that means coming up with a software solution and being able to code it up properly.

The hidden side of engineering is the **Process**, which means *how* we're actually building our product. Products don't just result from a single all-night coding session -- we need to make sure we're following a process that lets us create that Product in the most efficient and effective way possible.

****

AGILE DEVELOPMENT

1.15 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.

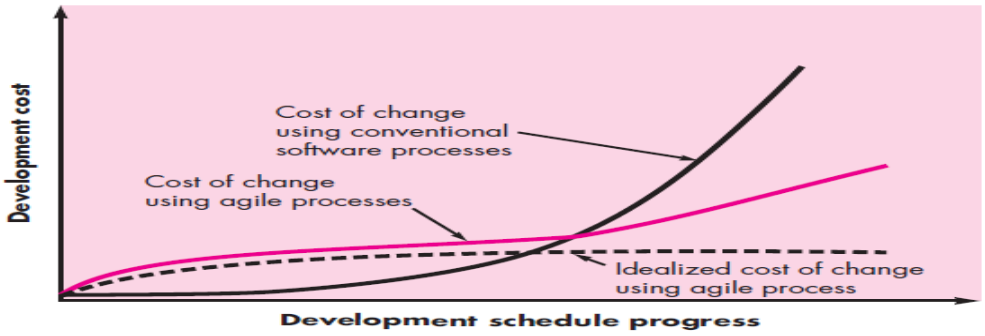
An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build 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.

****

1.16 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.



1.17AGILE 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

1.17.1 Agility Principles

Agility principles for those who want to achieve agility:

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

1.17.2 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 work schedule 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.

****

1.18 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.

1.18.1 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.

1.18.2 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.

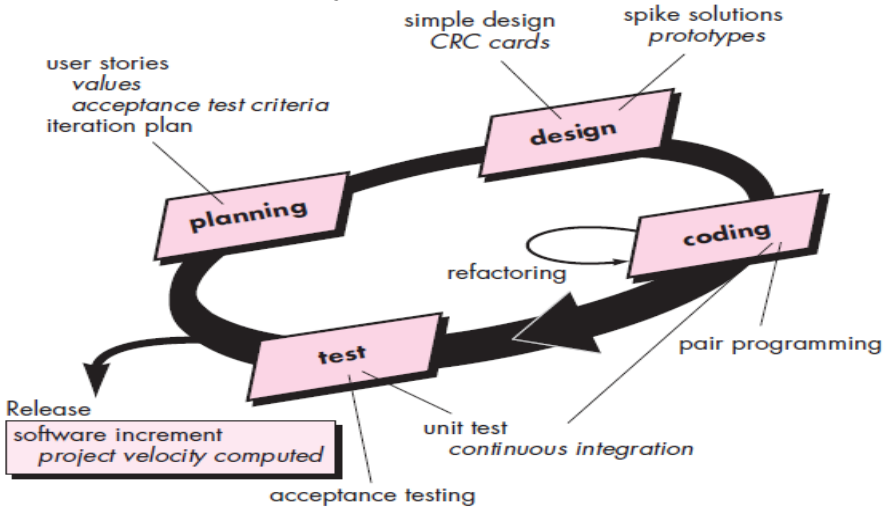


Fig : The Extreme Programming process

Key XP activities are

- **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.

1.18.3 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.

1.18.4 XP Debate

All new process models and methods spur worthwhile discussion and in some instances heated debate.
Among the issues that continue to trouble some critics of XP are:

• ***Requirements volatility***. Because the customer is an active member of the XP team, changes to requirements are requested informally. As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs. Proponents argue that this happens regardless of the process that is applied and that XP provides mechanisms for controlling scope creep.

• ***Conflicting customer needs***. Many projects have multiple customers, each with his own set of needs. In XP, the team itself is tasked with assimilating the needs of different customers, a job that may be beyond their scope of authority.

• ***Requirements are expressed informally***. User stories and acceptance tests are the only explicit manifestation of requirements in XP. Critics argue that a more formal model or specification is often needed to ensure that omissions, inconsistencies, and errors are uncovered before the system is built. Proponents counter that the changing nature of requirements makes such models and specification obsolete almost as soon as they are developed.

• ***Lack of formal design***. XP deemphasizes the need for architectural design and in many instances, suggests that design of all kinds should be relatively informal. Critics argue that when complex systems are built, design must be emphasized to ensure that the overall structure of the software will exhibit quality and maintainability. XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.

****

1.19 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)

1.19.1Adaptive 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.

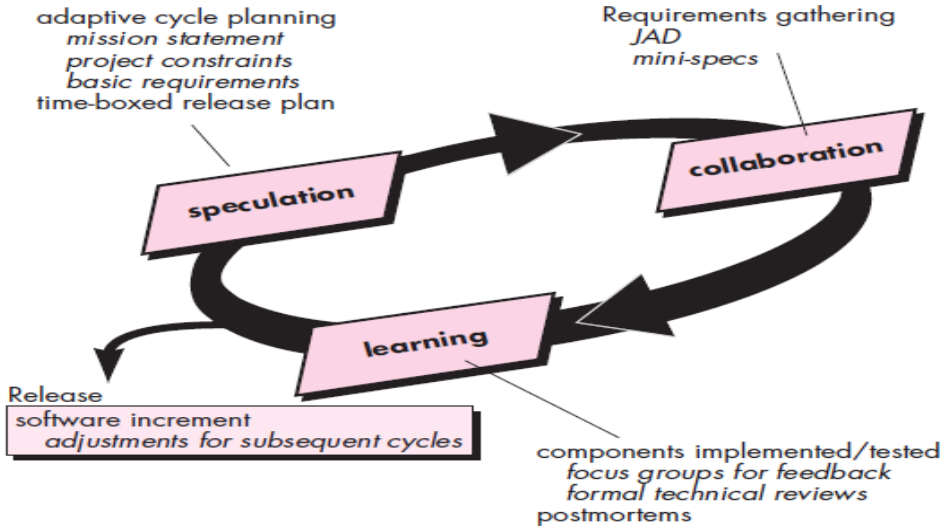


Fig : Adaptive software development

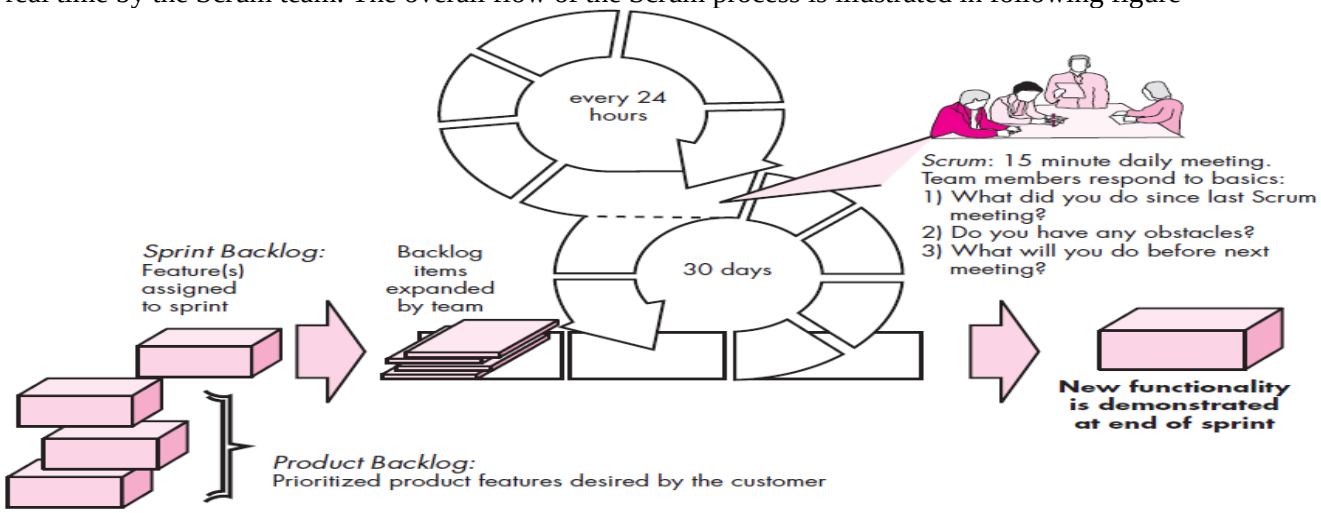
During ***speculation***, the project is initiated and ***adaptive cycle planning*** is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of 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**.

1.19.2 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.

1.19.3 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.

1.19.4 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.

1.19.5 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 improved Coad'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

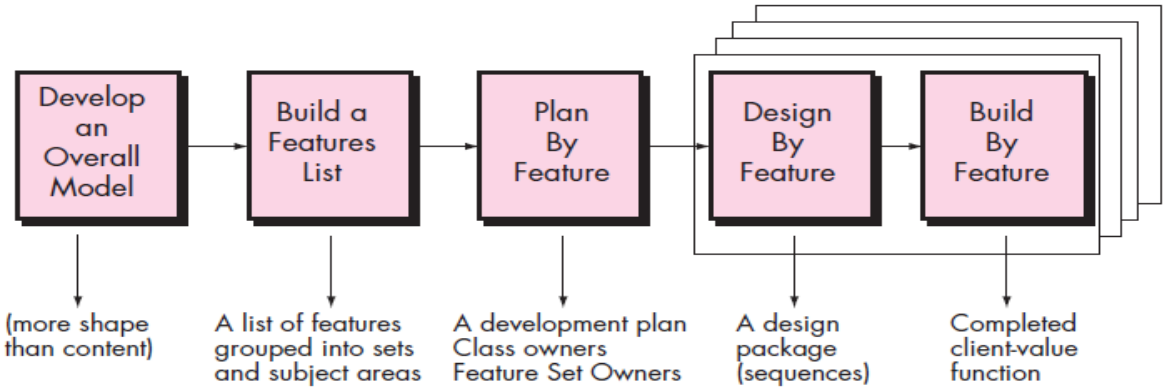


Fig : Feature Driven Development (FDD)

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**"

1.19.6 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.

1.19.7 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.

1.19.8 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.

UNIT- II

Understanding Requirements: Requirements Engineering, Establishing the groundwork, Eliciting Requirements, Developing Use Cases, Building the requirements model, Negotiating Requirements, Validating Requirements.
Requirements Modeling (Scenarios, Information and Analysis Classes): Requirements Analysis, Scenario-Based Modeling, UML Models that Supplement the Use Case, Data Modeling Concepts, Class-Based Modeling.
Requirements Modeling (Flow, Behavior, Patterns and WEBAPPS): Requirements Modeling Strategies, Flow-Oriented Modeling, Creating a Behavioral Model, Patterns for Requirements Modeling, Requirements Modeling for WebApps.

2.1 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.

****

2.2 ESTABLISHING THE GROUNDWORK

Requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team

2.2.1. Identifying Stakeholders

A ***stakeholder*** is anyone who has a direct interest in or benefits from the system that is to be developed. At inception, you should create a list of people who will contribute input as requirements are elicited..

2.2.2. Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. The information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

2.2.3. Working toward Collaboration

The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. It is, of course, the latter category that presents a challenge. Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

2.2.4 Asking the First Questions

Questions asked at the inception of the project should be "**context free**" . The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

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

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

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

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg call these "**meta-questions**" and propose the following list:

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

These questions will help to "**break the ice**" and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful.

****

2.3 ELICITING REQUIREMENTS

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

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.3.4 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.

****

2.4 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.

****

2.5 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..

2.5.1 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.

2.5.2 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.

****

2.6 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.

****

2.7 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.

****

2.8 REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:

- ***Scenario-based models*** of requirements from the point of view of various system "**actors**"
- ***Data models*** that depict the information domain for the problem
- ***Class-oriented models*** that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- ***Flow-oriented models*** that represent the functional elements of the system and how they transform data as it moves through the system
- ***Behavioral models*** that depict how the software behaves as a consequence of external "events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

Throughout requirements modeling, primary focus is on ***what,*** **not** ***how***. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?

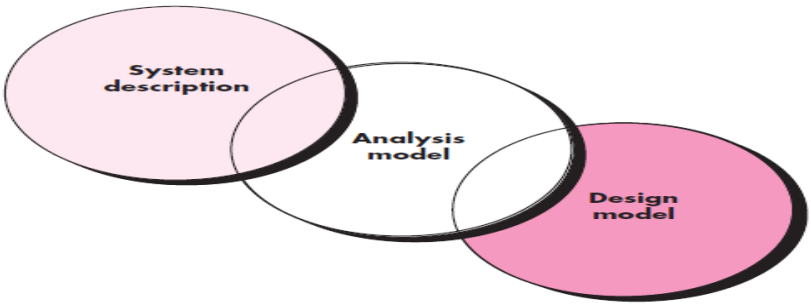


Fig : The requirements model as a bridge between the system description and the design model

The requirements model must achieve three primary objectives:

(1) To describe what the customer requires,

(2) to establish a basis for the creation of a software design, and

(3) to define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure.

2.8.1 Analysis Rules of Thumb

Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- ***The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.***
- ***Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.***
- ***Delay consideration of infrastructure and other nonfunctional models until design***. That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- ***Minimize coupling throughout the system***. It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.
- ***Be certain that the requirements model provides value to all stakeholders***. Each constituency has its own use for the model
- ***Keep the model as simple as it can be***. Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

2.8.2 Domain Analysis

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to identify common problem solving elements that are applicable to all applications within the domain, to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

2.8.3 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.

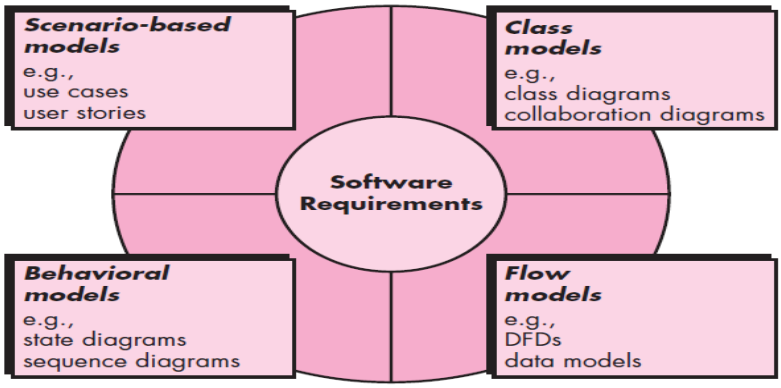


Fig : Elements of the analysis model

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

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

****

2.9 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.

2.9.1 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.

2.9.2 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.

2.9.3 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.

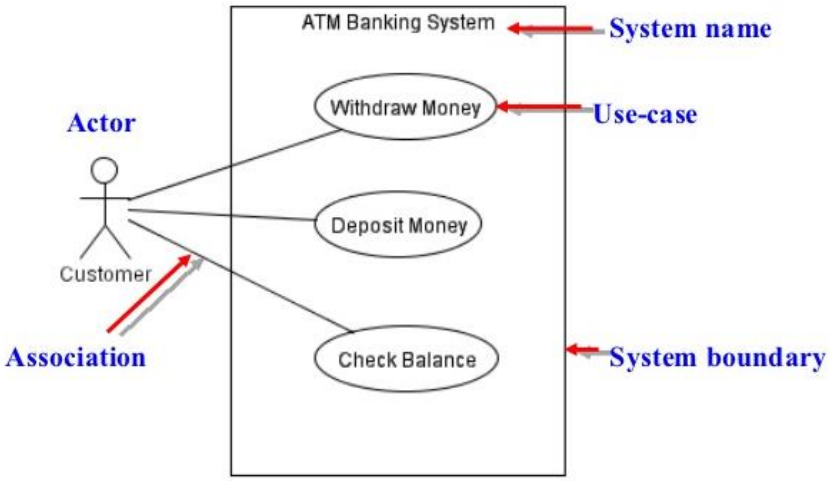


Fig : Simple Use Case Diagram

****

2.10 UML MODELS THAT SUPPLEMENT THE USE CASE

2.10.1 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.

2.10.2 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.

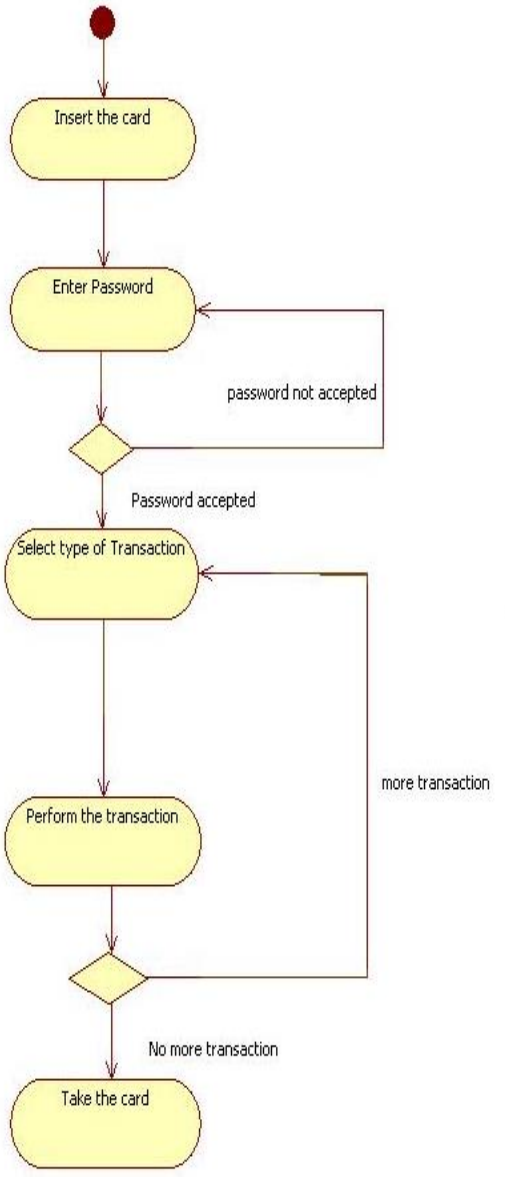
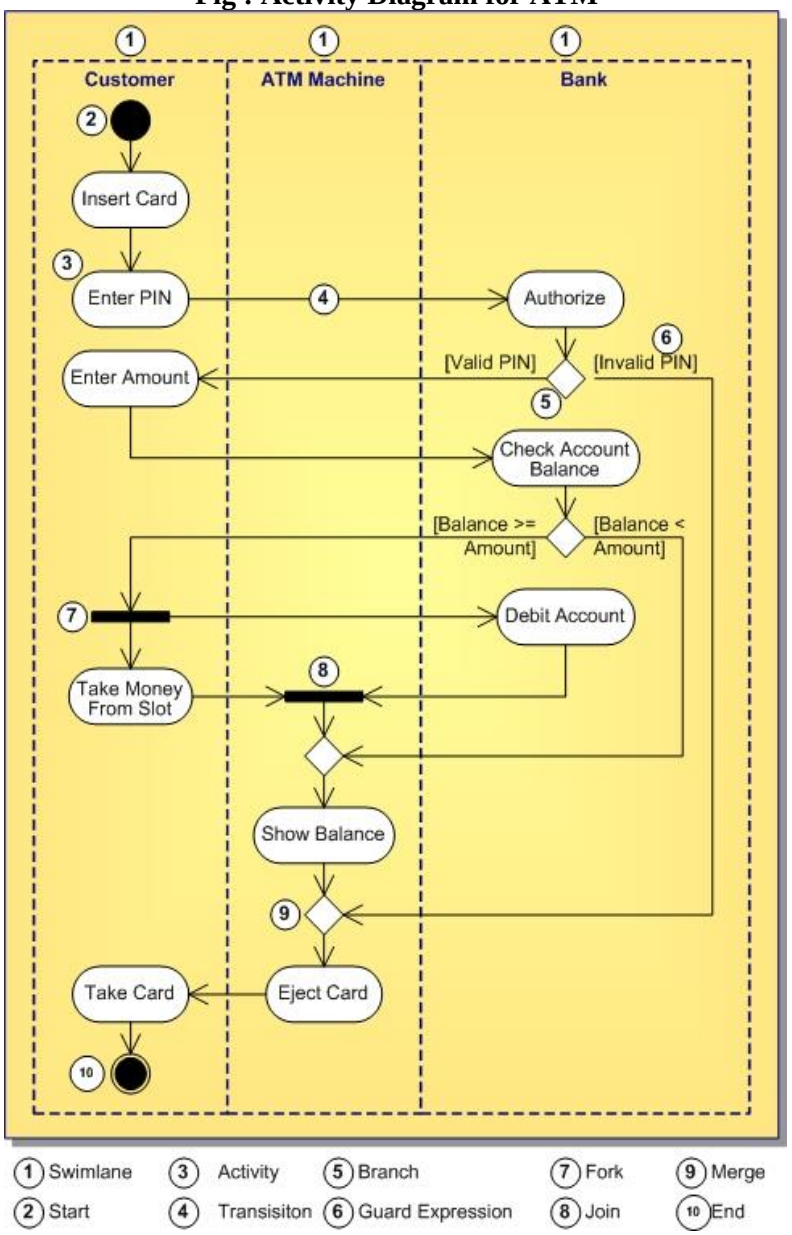


Fig : Activity Diagram for ATM

Fig : Activity Diagram for ATM



****

2.11 DATA MODELING CONCEPTS

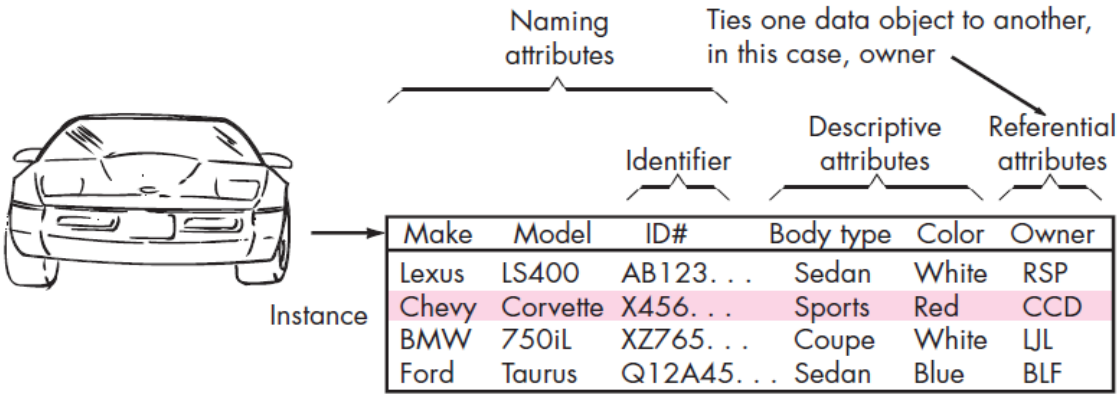
Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application. The most 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.

2.11.1 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.



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

Fig : Tabular representation of data objects

2.11.2 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.

2.11.3 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

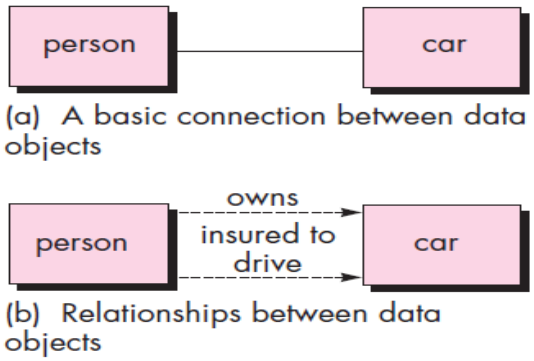


Fig : Relationships between data objects

****

2.12 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.

2.12.1 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.

2.12 .2 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.

2.12.3 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.

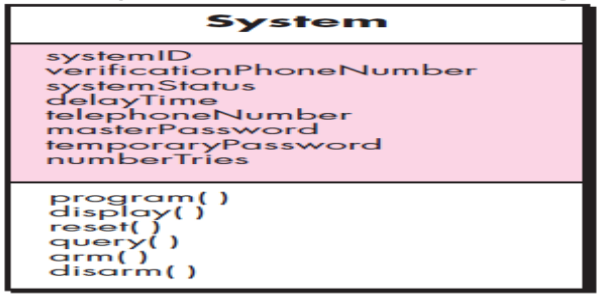


Fig : Class diagram for the system class

2.12.4 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.

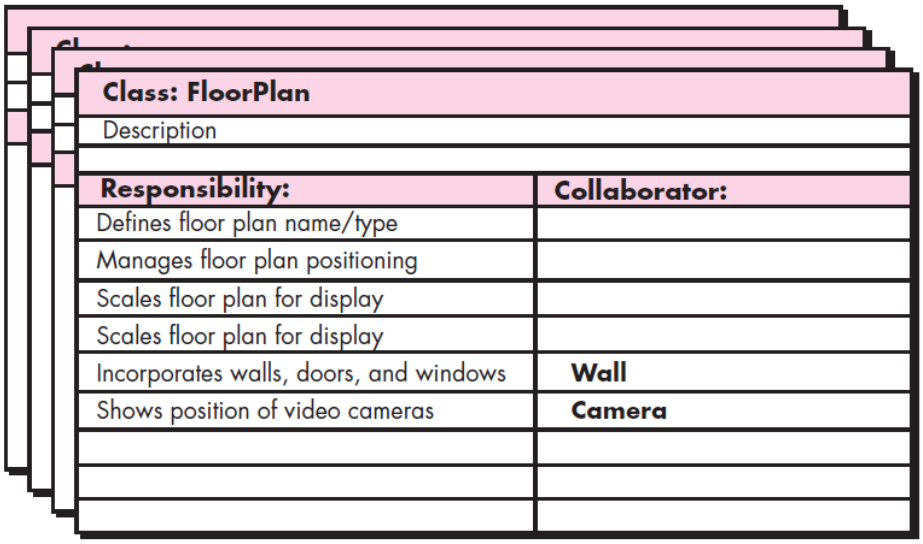


Fig : A CRC model index card

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.

2.12.5 Associations and Dependencies

An ***association*** defines a relationship between classes. An association may be further defined by indicating ***multiplicity***. **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**>>).

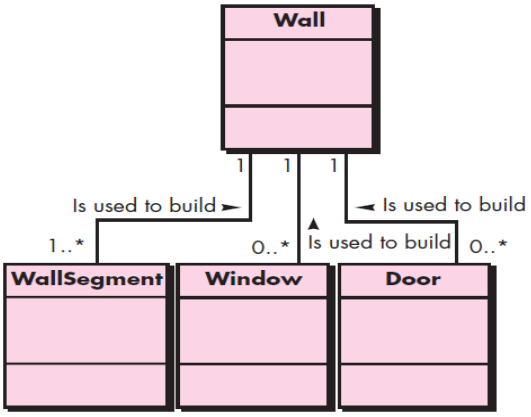


Fig : Multiplicity

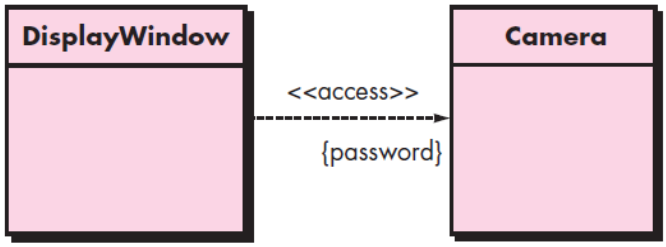


Fig : Dependencies

2.12.6 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.

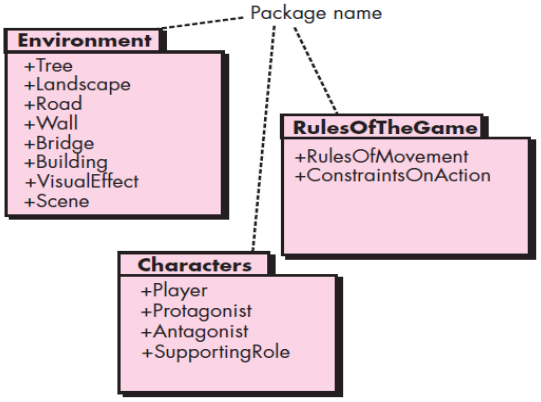


Fig : Packages

****

2.13 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.

****

2.14 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.

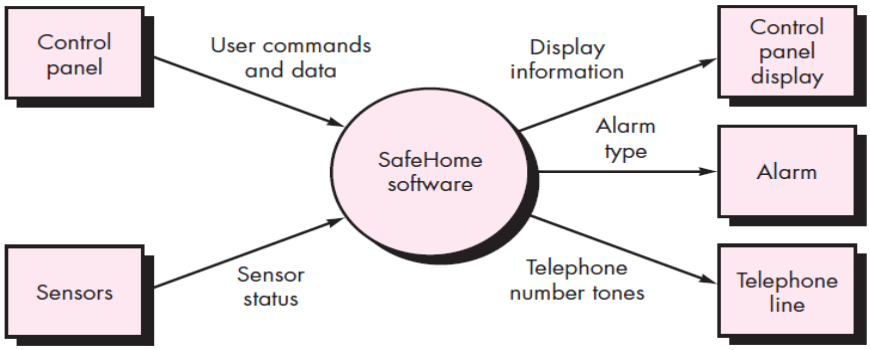


Fig : Context-level DFD for the Safe Home security function

2.14.1Creating 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**."

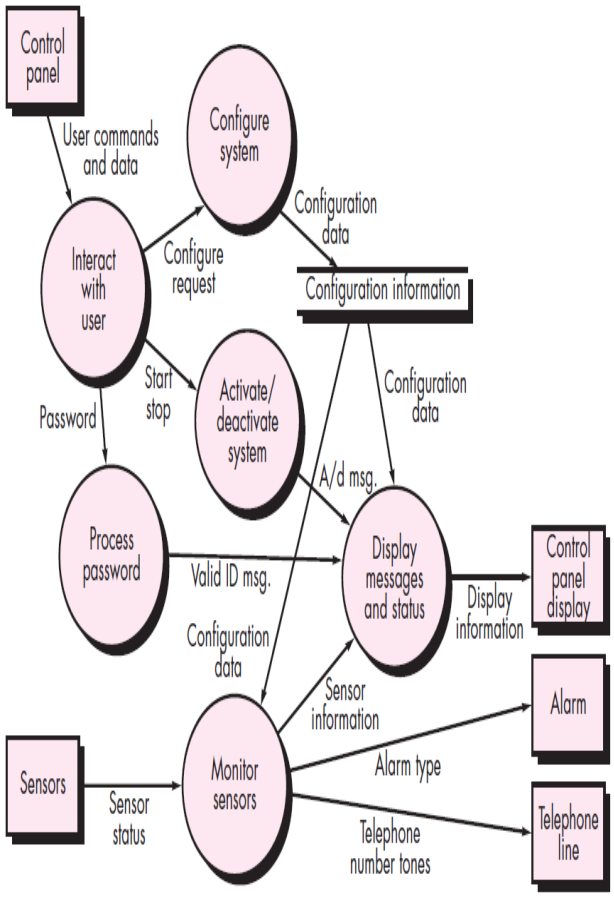


Fig: Level 1 DFD for SafeHome security function

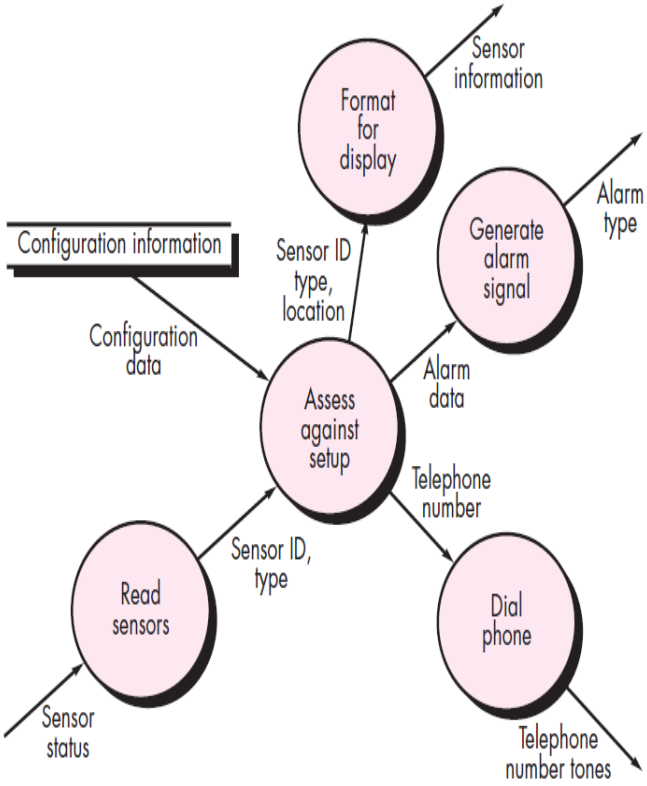


Fig : Level 2 DFD that refines the monitor sensors process

2.14.2. 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;

2.14.3 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.

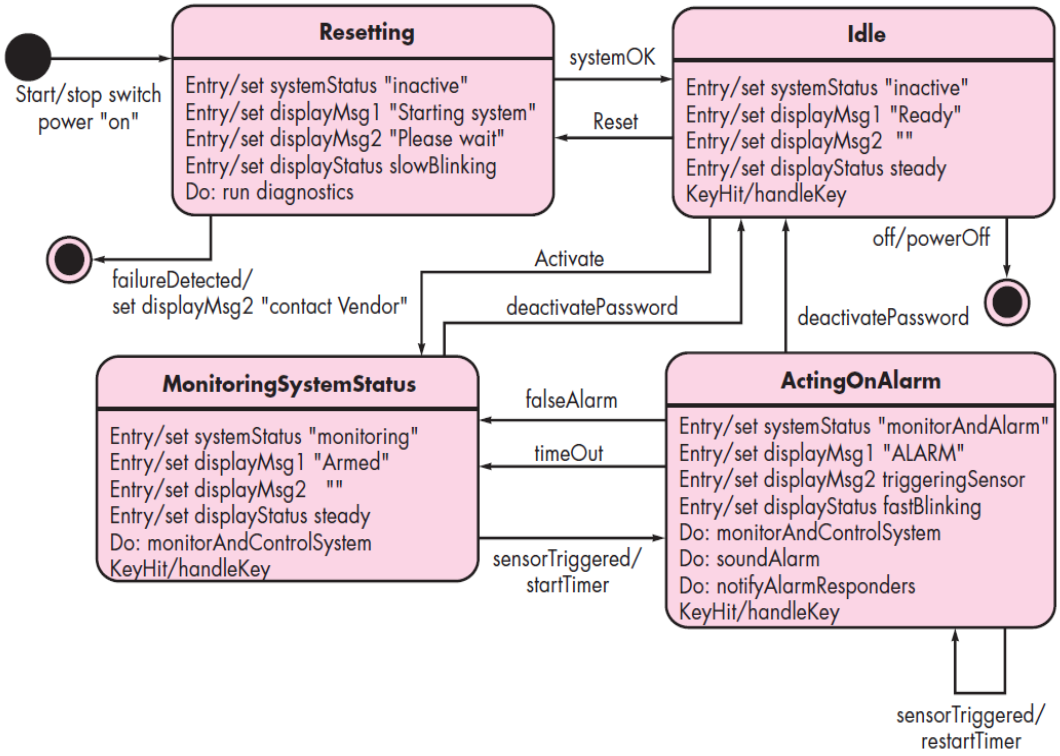


Fig : State diagram for SafeHome security function

2.14.4 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.

****

2.15 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.

2.15.1 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 .

2.15.2 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

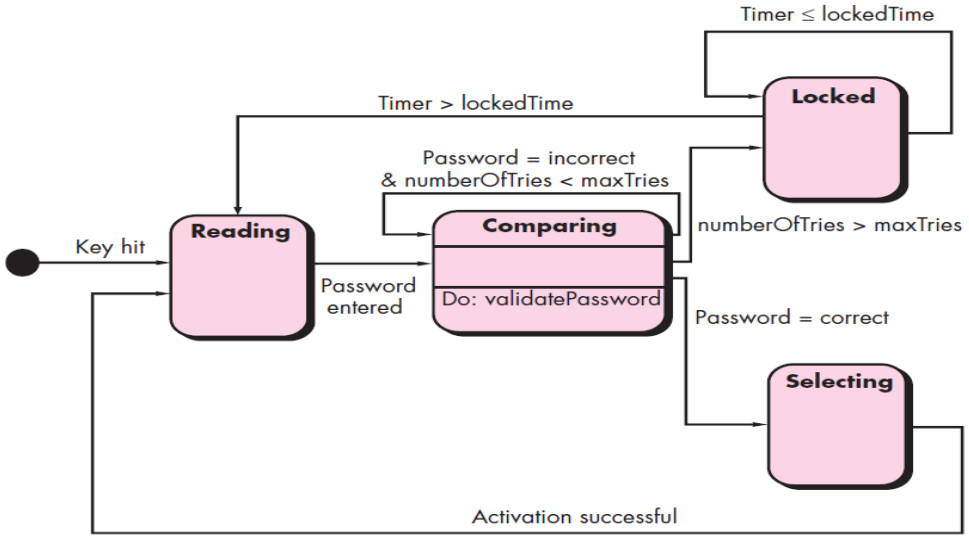


Fig : State diagram for the Control Panel class

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.

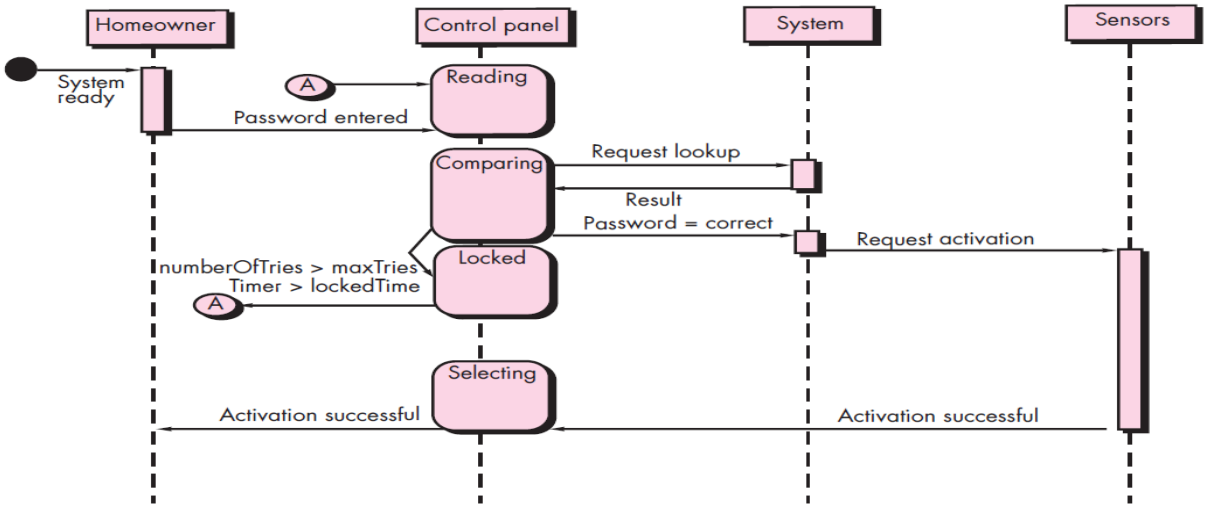


Fig : Sequence diagram (partial) for the *SafeHome* security function

****

2.16 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"

****

2.17 REQUIREMENTS MODELING FOR WEBAPPS

Requirements analysis does take time, but solving the wrong problem takes even more time.

2.17.1 How Much Analysis Is Enough?

The degree to which requirements modeling for WebApps is emphasized depends on the following factors:

- Size and complexity of WebApp increment.
- Number of stakeholders
- Size of the WebApp team.
- Degree to which members of the WebApp team have worked together
- Degree to which the organization's success is directly dependent on the success of the design of a specific part of the WebApp.

It only demands an analysis of those requirements that affect only that part of the WebApp.

2.17.2 Requirements Modeling Input

The requirements model provides a detailed indication of the true structure of the problem and provides insight into the shape of the solution. Requirements analysis refines this understanding by providing additional interpretation. As the problem structure is delineated as part of the requirements model.

2.17.3 Requirements Modeling Output

Requirements analysis provides a disciplined mechanism for representing and evaluating WebApp content and function, the modes of interaction that users will encounter, and the environment and infrastructure in which the WebApp resides. Each of these characteristics can be represented as a set of models that allow the WebApp requirements to be analyzed in a structured manner. While the specific models depend largely upon the nature of the WebApp, there are **five** main classes of models:

- **Content model**—identifies the full spectrum of content to be provided by the WebApp. Content includes text, graphics and images, video, and audio data.
- **Interaction model**—describes the manner in which users interact with the WebApp.
- **Functional model**—defines the operations that will be applied to WebApp content and describes other processing functions that are independent of content but necessary to the end user.
- **Navigation model**—defines the overall navigation strategy for the WebApp.
- **Configuration model**—describes the environment and infrastructure in which the WebApp resides.

2.17. 4. Content. Model for WebApps

The content model contains structural elements that provide an important view of content requirements for a WebApp. These structural elements encompass content objects and all analysis classes, user-visible entities that are created or manipulated as a user interacts with the

Content can be developed prior to the implementation of the WebApp, while the WebApp is being built, or long after the WebApp is operational.

A *content object* might be a textual description of a product, an article describing a news event, an action photograph taken at a sporting event, a user's response on a discussion forum, an animated representation of a corporate logo, a short video of a speech, or an audio overlay for a collection of presentation slides. The content objects might be stored

as separate files, embedded directly into Web pages, or obtained dynamically from a database. Content objects can be determined directly from use cases by examining the scenario description for direct and indirect references to content. The content model must be capable of describing the content object **Component.**

2.17.5. Interaction Model for WebApps

Interaction model that can be composed of one or more of the following elements: (1) use cases, (2) sequence diagrams, (3) state diagrams,16 and/or (4) user interface prototypes.

2.17.6. Functional Model for WebApps

The *functional model* addresses two processing elements of the WebApp, each representing a different level of procedural abstraction: (1) user-observable functionality that is delivered by the WebApp to end users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.
User-observable functionality encompasses any processing functions that are initiated directly by the user.

2.17.7 Configuration Models for WebApps

The configuration model is nothing more than a list of server-side and client-side attributes. However, for more complex WebApps, a variety of configuration complexities may have an impact on analysis and design. The UML deployment diagram can be used in situations in which complex configuration architectures must be considered.

2.17.8 Navigation Modeling

Navigation modeling considers how each user category will navigate from one WebApp element (e.g., content object) to another. The mechanics of navigation are defined as part of design. At this stage, you should focus on overall navigation requirements. The following questions should be considered:

- Should certain elements be easier to reach than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?
- Should a full navigation map or menu be available at every point in a user's interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user "store" his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? Overlaying the existing browser window? As a new browser window? As a separate frame?

These and many other questions should be asked and answered as part of navigation analysis.

UNIT- III

Design Concepts: Design with Context of Software Engineering, The Design Process, Design Concepts, The Design Model.
Architectural Design: Software Architecture, Architecture Genres, Architecture Styles, Architectural Design, Assessing Alternative Architectural Designs, Architectural Mapping Using Data Flow.
Component-Level Design: Component, Designing Class-Based Components, Conducting Component-level Design, Component Level Design for WebApps, Designing Traditional Components, Component-Based Development.

Design Concepts

Introduction: Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides you in the design work you must perform. Design is pivotal to successful software engineering.
The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight Software design changes continually as new methods, better analysis, and broader understanding evolve.

3.1 DESIGN WITH CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).
Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in following figure.
The requirements model, manifested by **scenario-based, class-based, flow-oriented, and behavioral elements,** feed the design task.
The **data/class design** transforms class models into design class realizations and the requisite data structures required to implement the software.
The **architectural design** defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

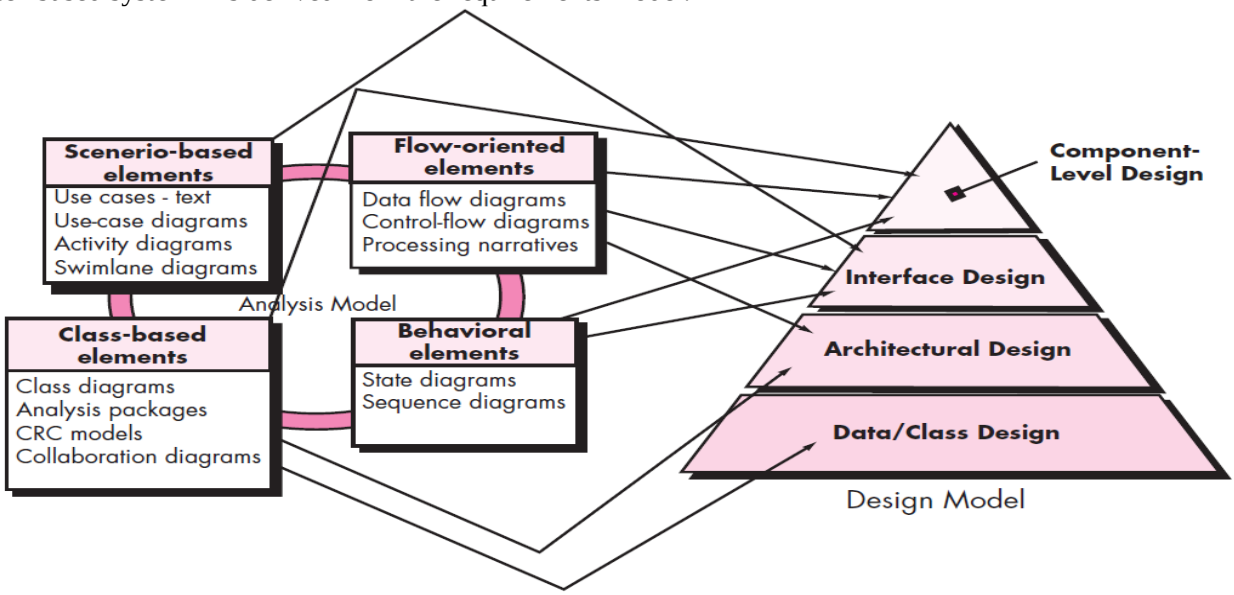


Fig : Translating the requirements model into the design model

The **interface design** describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The **component-level design** transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

The importance of software design can be stated with a single word—***quality.*** Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow.

****

3.2 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**

3.2.1 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.

3.2.2 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.
****

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 Separation of Concerns

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

3.3.5 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

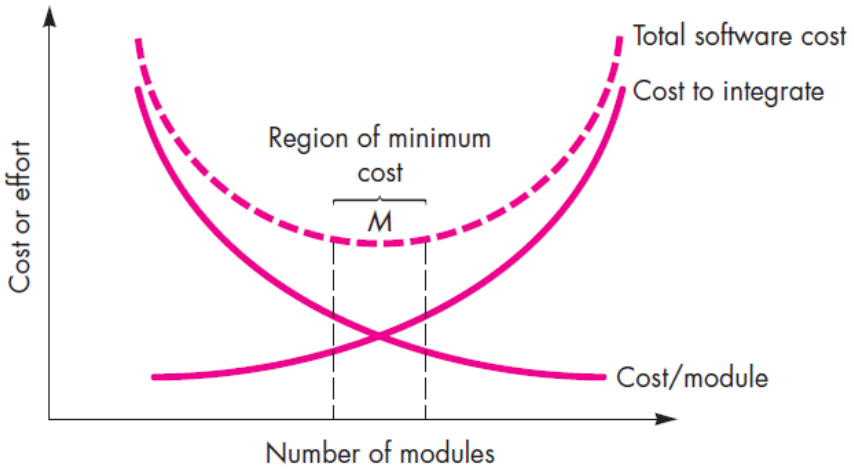


Fig: Modularity and software cost

3.3.6 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.

3.3.7 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**.

3.3.8 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.

3.3.9 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.

3.3.10 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."**

3.3.11 Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. OO
design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

3.3.12 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.

****

3.4 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.

3.4.1. Data Design Elements

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

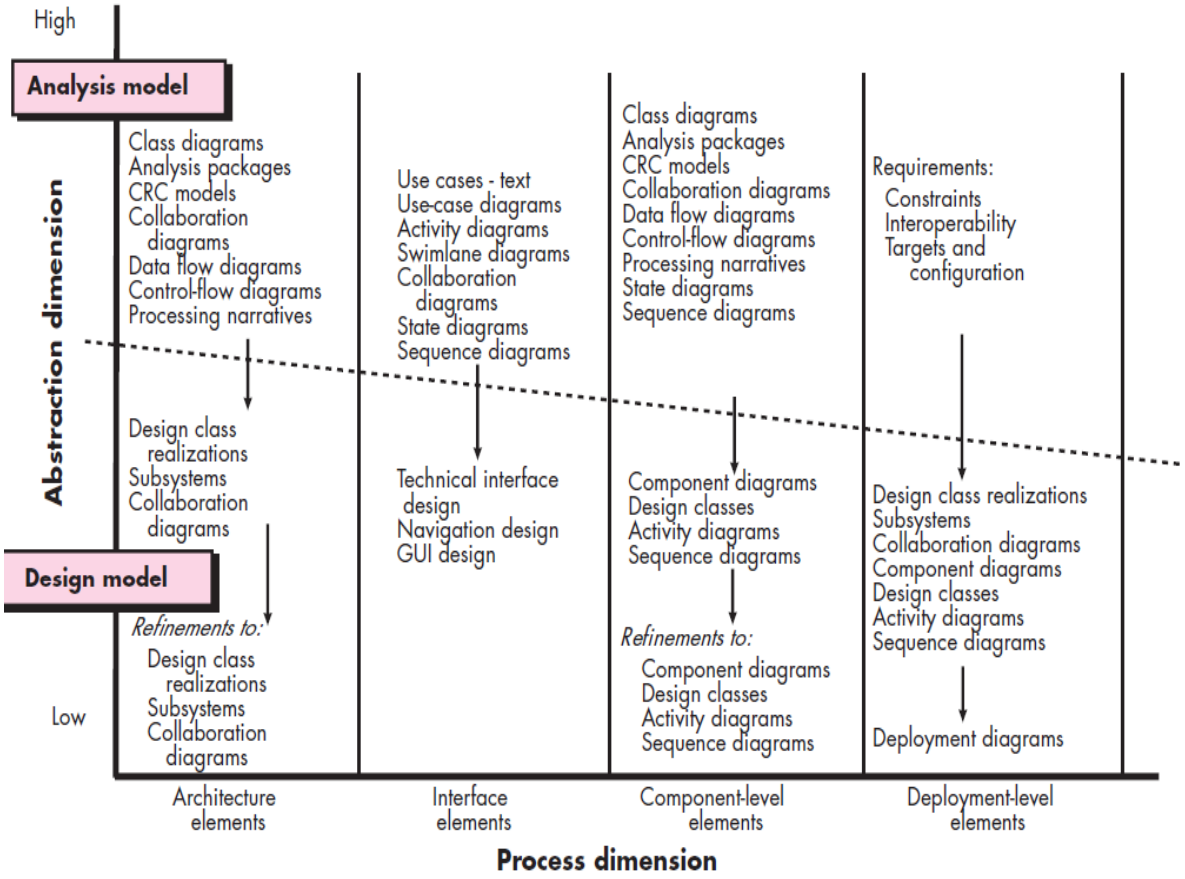


Fig : Dimensions of the design model

3.4.2 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.

3.4.3. Interface Design Elements

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

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

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

3.4.4 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.

3.4.5 Deployment-Level Design Elements

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

****

3.5 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.

3.5.1 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.

3.5.2 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.

3.5.3 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."

3.5.4 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.

****

3.6 ARCHITECTURE GENRES

The architectural ***genre*** will often dictate the specific architectural approach to the structure that must be built. In the context of architectural design, *genre* implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. **Grady Booch** suggests the following architectural genres for software-based systems:

• **Artificial intelligence**—Systems that simulate or augment human cognition, locomotion, or other organic processes.
• **Commercial and nonprofit**—Systems that are fundamental to the operation of a business enterprise.
• **Communications**—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
• **Content authoring**—Systems that are used to create or manipulate textual or multimedia artifacts. • **Devices**—Systems that interact with the physical world to provide some point service for an individual.
• **Entertainment and sports**—Systems that manage public events or that provide a large group entertainment experience.
• **Financial**—Systems that provide the infrastructure for transferring and managing money and other securities.
• **Games**—Systems that provide an entertainment experience for individuals or groups.
• **Government**—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
• **Industrial**—Systems that simulate or control physical processes.
• **Legal**—Systems that support the legal industry.
• **Medical**—Systems that diagnose or heal or that contribute to medical research.
• **Military**—Systems for consultation, communications, command, control, and intelligence as well as offensive and defensive weapons.
• **Operating systems**—Systems that sit just above hardware to provide basic software services.
• **Platforms**—Systems that sit just above operating systems to provide advanced services.
• **Scientific**—Systems that are used for scientific research and applications.
• **Tools**—Systems that are used to develop other systems.
• **Transportation**—Systems that control water, ground, air, or space vehicles.
• **Utilities**—Systems that interact with other software to provide some point service.

****

3.7 ARCHITECTURE 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.

3.7.1 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.***

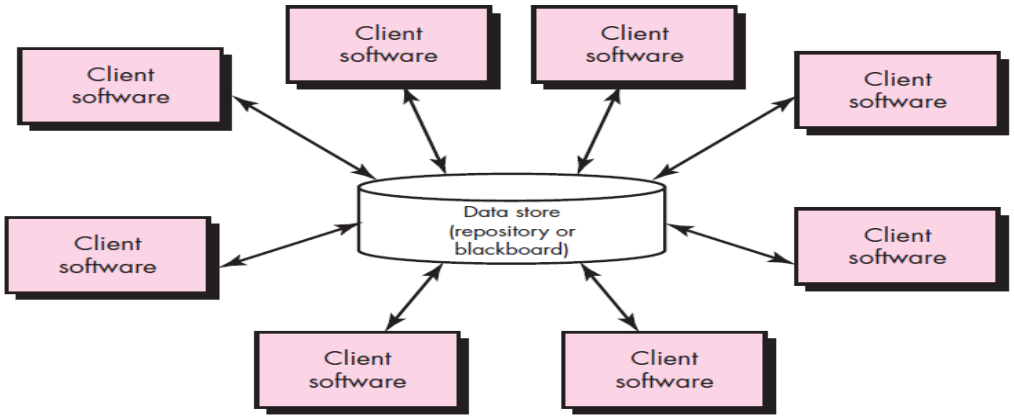


Fig : Data-centered architecture

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.

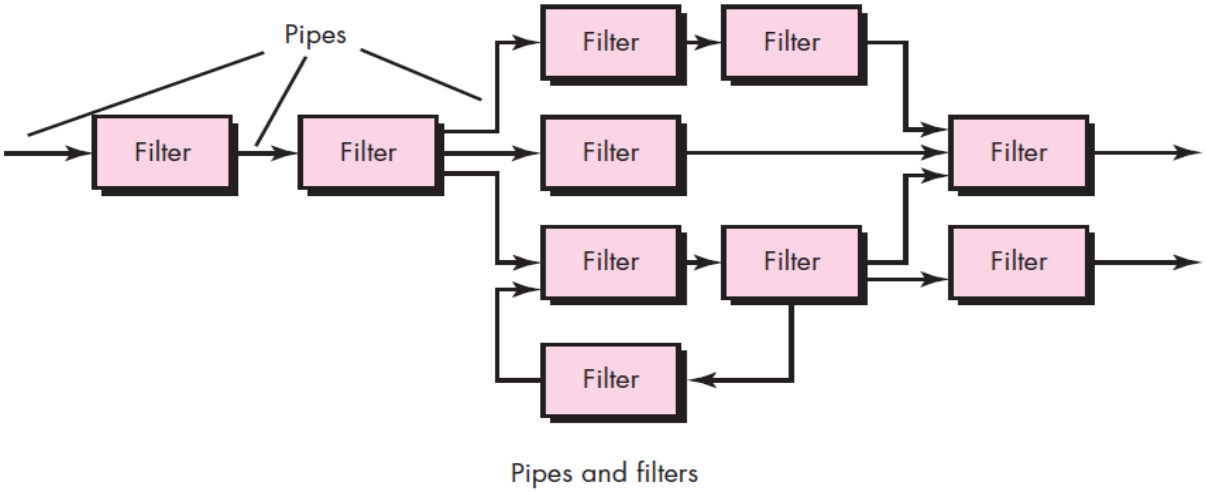


Fig : Data-flow architecture

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.

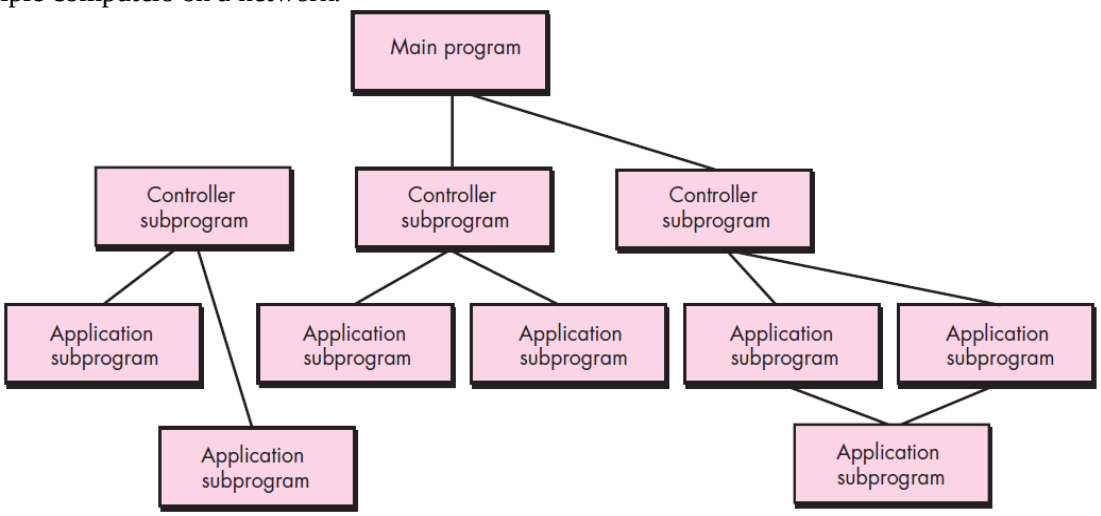


Fig : Main program/subprogram architecture

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.

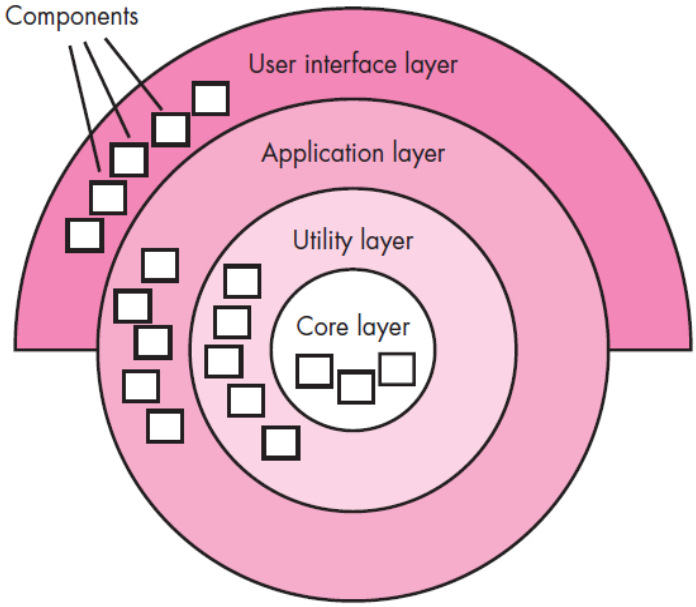


Fig : Layered architecture

3.7.2 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.

3.7.3 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.

****

3.8 ARCHITECTURAL DESIGN

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

3.8.1 Representing the System in Context

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

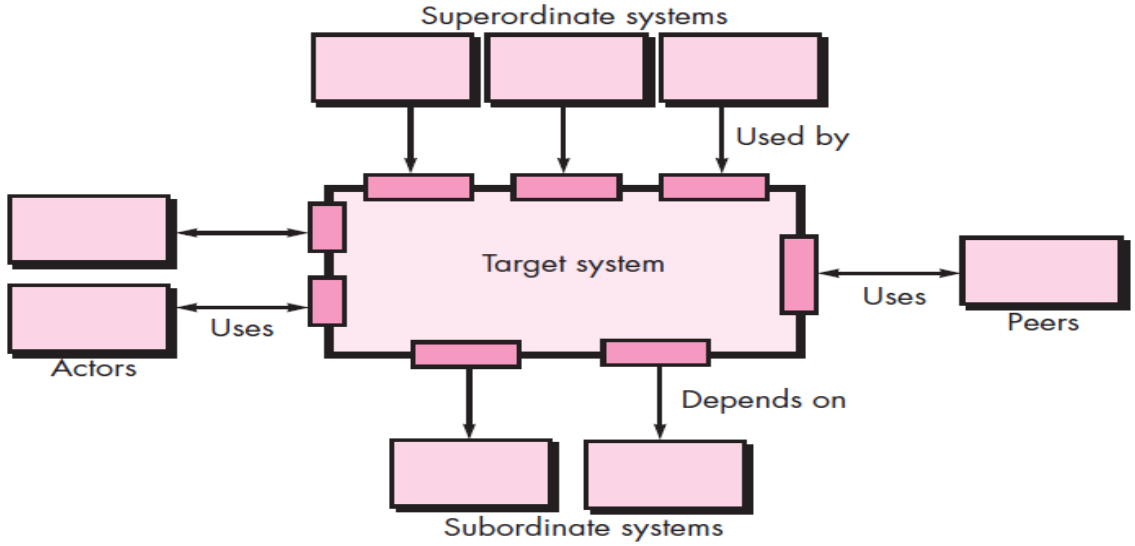


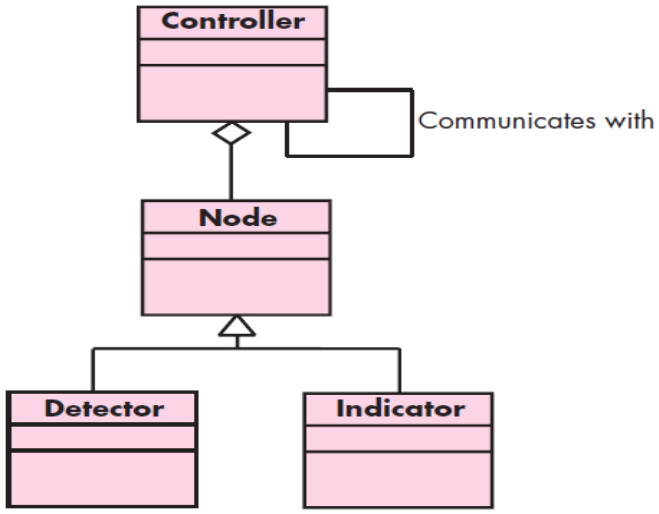
Fig : Architectural context diagram

3.8.2 Defining Archetypes

An ***archetype*** is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

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.



3.8.3 Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. The 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.

****

3.9 ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

3.9.1 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***.

3.9.2 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.

3.9.3 Architectural Description Languages

Architectural description language (ADL) provides a semantics and syntax for describing 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.
****

3.10 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.

3.10.1 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.

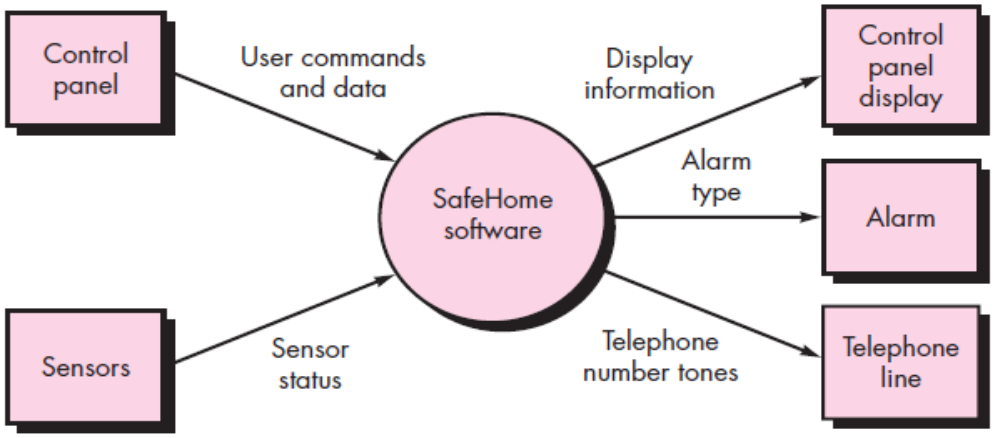


Fig : Context-level DFD for the *SafeHome* security function

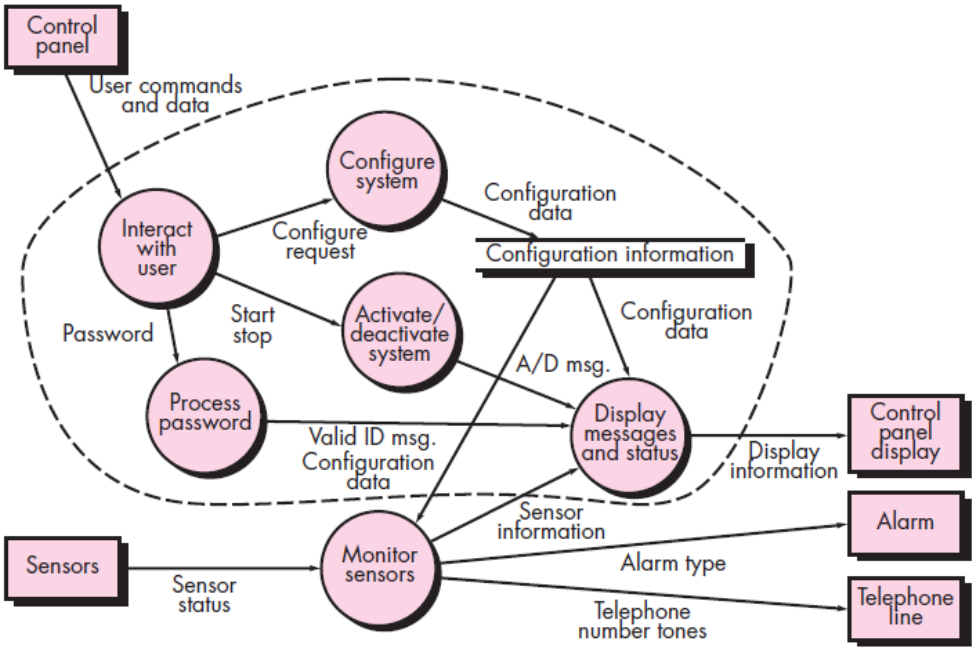


Fig : Level 1 DFD for the *SafeHome* security function

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.

3.10.2 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.

****

3.11 COMPONENT, DESIGNING CLASS-BASED COMPONENTS

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.

3.11.1 An Object-Oriented View

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

3.11.2 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

3.11.3 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.

3.11.4 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.

3.11.5 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.

3.11.6 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 package or 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**.
****

3.12 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.

****

3.13 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.

3.13.1 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.

3.13.2 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.

****

3.14 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. 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.

3.14.1 Graphical Design Notation

The following figure illustrates **three** structured constructs.

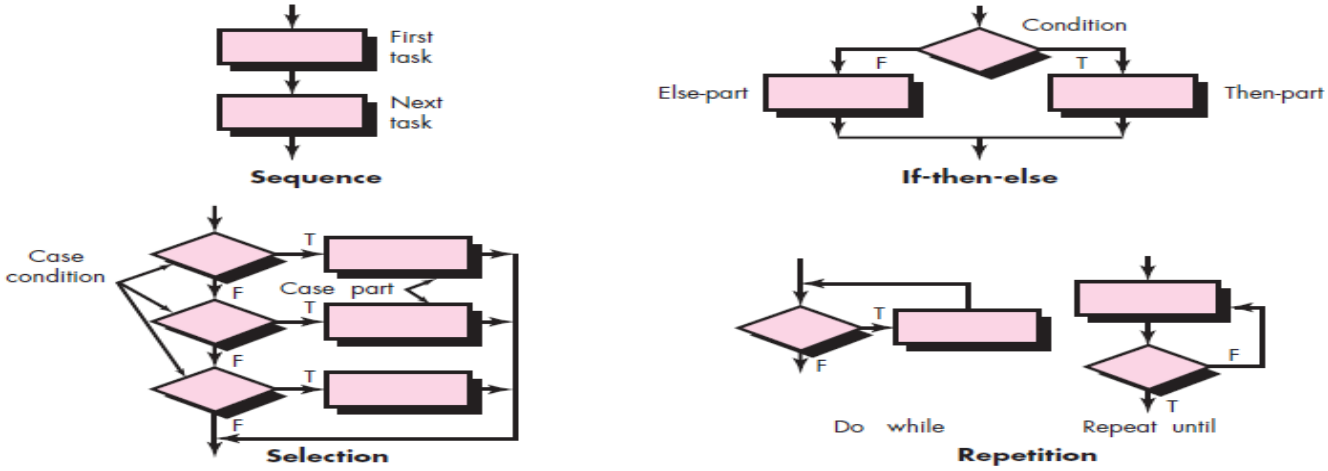


Fig : Flowchart constructs

”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 ***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***.

3.14.2 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.

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

Fig : Decision table

3.14.3 Program Design Language

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

3.15 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."

3.15.1 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.

3.15.2 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 : An 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.

3.15.3 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.

3.15.4 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 implementation.

UNIT- IV

User Interface Design: The Golden Rules, User Interface Analysis and Design, Interface Analysis, Interface Design Steps, WebApp Interface Design, Design Evaluation.

WebApp Design: WebApp Design Quality, Design Goal, A Desigin Pyramid for WebApps, WebApp Interface Design, Aestheic Design, Content Design, Architecture Design, Navigation Design, Component-Level Design, Object-Oriented Hypermedia Design Method(OOHMD).

4.1 THE GOLDEN RULES

The Mandel coins **three** *golden rules:*

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

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

4.1.1 Place the User in Control

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

- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions.** An interaction mode is the current state of the interface.
- **Provide for flexible interaction.** Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.
- **Allow user interaction to be interruptible and undoable.** Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.
- **Streamline interaction as skill levels advance and allow the interaction to be customized.** Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.
- **Hide technical internals from the casual user.** The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.
- **Design for direct interaction with objects that appear on the screen.** The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

4.1.2 Reduce the User's Memory Load

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

- **Reduce demand on short-term memory.** When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.
- **Establish meaningful defaults.** The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.
- **Define shortcuts that are intuitive.** When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).
- **The visual layout of the interface should be based on a real-world metaphor.** For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process.

This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

- **Disclose information in a progressive fashion.** The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

4.1.3 Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented.

Mandel defines a set of design principles that help make the interface consistent:

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

****

4.2 USER INTERFACE ANALYSIS AND DESIGN

4.2.1 Interface Analysis and Design Models

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

The user model establishes the profile of end users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality". Users can be categorized as:

Novices. No syntactic knowledge1 of the system and little semantic knowledge of the application or computer usage in general.

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

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

The user's ***mental model*** (system perception) is the image of the system that end users carry in their heads.

The ***implementation model*** combines the outward manifestation of the computer based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively.

These models enable the interface designer to satisfy a key element of the most important principle of user interface design: "**Know the user, know the tasks**."

4.2.2 The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model Referring to the following figure, the user interface analysis and design process begins at the interior of the spiral and encompasses **four** distinct framework activities

(1) interface analysis and modeling,

(2) interface design,

(3) interface construction, and

(4) interface validation.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. Once general requirements have been defined, a more detailed ***task analysis*** is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

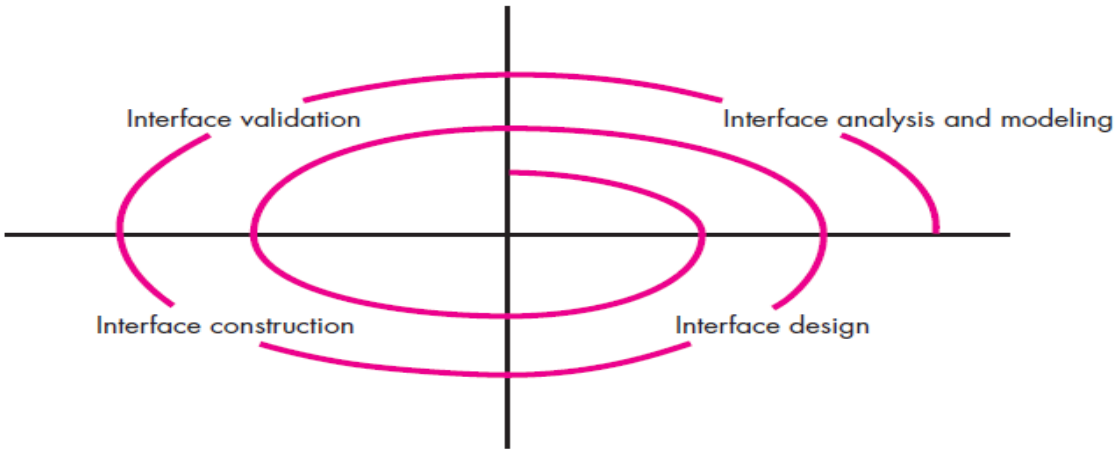


Fig : The user interface design process

Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

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

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

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

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

****

4.3 INTERFACE ANALYSIS

A key tenet of all software engineering process models is this: ***understand the problem before you attempt to design a solution***. In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. These elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

4.3.1 User Analysis

The phrase "user interface" is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Information from a broad array of sources can be used to accomplish this:

- **User Interviews.** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.
- **Sales input.** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

- **Marketing input.** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.
- **Support input.** Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions will help you to better understand the users of a system:
• Are users trained professionals, technicians, clerical, or manufacturing workers?
• What level of formal education does the average user have?
• Are the users capable of learning from written materials or have they expressed a desire for classroom training?
• Are users expert typists or keyboard phobic?
• What is the age range of the user community?
• Will the users be represented predominately by one gender?
• How are users compensated for the work they perform? • Do users work normal office hours or do they work until the job is done?
• Is the software to be an integral part of the work users do or will it be used only occasionally?
• What is the primary spoken language among users?
• What are the consequences if a user makes a mistake using the system?
• Are users experts in the subject matter that is addressed by the system?
• Do users want to know about the technology that sits behind the interface?
Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

4.3.2 Task Analysis and Modeling

The goal of task analysis is to answer the following questions:
• What work will the user perform in specific circumstances?
• What tasks and subtasks will be performed as the user does the work?
• What specific problem domain objects will the user manipulate as work is performed?
• What is the sequence of work tasks—the workflow?
• What is the hierarchy of tasks?

4.3.3 Analysis of Display Content

Analysis of display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files). These data objects may be (1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.
During this interface analysis step, the format and aesthetics of the content are considered. Among the questions that are asked and answered are:
• Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?
• Can the user customize the screen location for content?
• Is proper on-screen identification assigned to all content?
• If a large report is to be presented, how should it be partitioned for ease of understanding?
• Will mechanisms be available for moving directly to summary information for large collections of data?
• Will graphical output be scaled to fit within the bounds of the display device that is used?
• How will color be used to enhance understanding?
• How will error messages and warnings be presented to the user?
The answers to these (and other) questions will help you to establish requirements.

4.3.4 Analysis of the Work Environment

Hackos and Redish discuss the importance of work environment analysis when they state: *eople do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the*

type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use.
In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.
****

4.4 INTERFACE DESIGN STEPS

Although many different user interface design models have been proposed, all suggest some combination of the following steps:
1. Using information developed during interface analysis, define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

4.4.1 Applying Interface Design Steps

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.
Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified.

4.4.2 User Interface Design Patterns

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. A design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem**.**

4.4.3 Design Issues

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

- **Response time.** System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action until the software responds with desired output or action. System response time has **two** important characteristics: **length and variability**. If system response is too **long**, user frustration and stress are inevitable. ***Variability*** refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long.
- **Help facilities.** Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick.

A number of design issues must be addressed when a help facility is considered:
• Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.
• How will the user request help? Options include a help menu, a special function key, or a HELP command.
• How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.
• How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.
• How will help information be structured?

- **Error handling.** Error messages and warnings are **"bad news**" delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration. In general, every error message or warning produced by an interactive system should have the following characteristics:

• The message should describe the problem in jargon that the user can understand.
• The message should provide constructive advice for recovering from the error.

• The message should indicate any negative consequences of the error so that the user can check to ensure that they have not occurred
• The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."
• The message should be "nonjudgmental." That is, the wording should never
place blame on the user.

- **Menu and command labeling.** The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

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

Application accessibility.: . *Accessibility* for users who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines many designed for Web applications but often applicable to all types of software provide detailed suggestions for designing interfaces that achieve varying levels of accessibility.

Internationalization. Interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software**.** ***Localization*** features enable the interface to be customized for a specific market.

A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard [Uni03] has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

****

4.5 WEBAPP INTERFACE DESIGN

Dix argues that you should design a WebApp interface so that it answers **three** primary questions or the end user:

Where am I? The interface should (1) provide an indication of the WebApp that has been accessed and (2) inform the user of her location in the content hierarchy.

What can I do now? The interface should always help the user understand his current options like what functions are available, what links are live, what content is relevant?

Where have I been, where am I going? The interface must facilitate navigation. Hence, it must provide a "map" of where the user has been and what paths may be taken to move elsewhere within the WebApp.

An effective WebApp interface must provide answers for each of these questions as the end user navigates through content and functionality.

4.5.1 Interface Design Principles and Guidelines

A good WebApp interface is understandable and forgiving, providing the user with a sense of control. Bruce Tognozzi defines a set of fundamental characteristics that all interfaces should exhibit and in doing so, establishes a philosophy that should be followed by every WebApp interface designer.

Effective interfaces are visually apparent and forgiving, instilling in their users a sense of
control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.

Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.

Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

In order to design WebApp interfaces that exhibit these characteristics, Tognozzi identifies a set of overriding design principles:

- **Anticipation.** *A WebApp should be designed so that it anticipates the user's next move.*
- **Communication.** *The interface should communicate the status of any activity initiated by the user.* Communication can be obvious or subtle. The interface should also communicate user status (e.g., the user's identification) and her location within the WebApp content hierarchy.
- **Consistency.** *The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout the WebApp.*
- **Controlled autonomy.** *The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.*
- **Efficiency.** *The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client server environment that executes it.*
- **Flexibility.** *The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the WebApp in a somewhat random fashion.* In every case, it should enable the user to understand where he is and provide the user with functionality that can undo mistakes and retrace poorly chosen navigation paths.
- **Focus.** *The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.*
- **Fitt's law.** *"The time to acquire a target is a function of the distance to and size of the target".* Fitt's law "is an effective method of modeling rapid, aimed movements, where one appendage starts at rest at a specific start position, and moves to rest within a target area". If a sequence of selections or standardized inputs is defined by a user task, the first selection (e.g., mouse pick) should be physically close to the next selection.
- **Human interface objects.** *A vast library of reusable human interface objects has been developed for WebApps. Use them.*
- **Latency reduction.** *Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the WebApp should use multitasking in a way that lets the user proceed with work as if the operation has been completed.* In addition to reducing latency, delays must be acknowledged so that the userunderstands what is happening. This includes (1) providing audio feedback when a selection does not result in an immediate action by the WebApp, (2) displaying an animated clock or progress bar to indicate that processing is under way, and (3) providingsome entertainment (e.g., an animation or text presentation) while lengthy processing occurs.
- **Learnability.** *A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.* In general the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to the user.
- **Metaphors.** *An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user. Metaphors are an excellent idea because they mirror real-world experience. Just be sure that the metaphor you choose is well known to end users.*
- **Maintain work product integrity.** *A work product (e.g., a form completed by the user, a user-specified list) must be automatically saved so that it will not be lost if an error occurs.*
- **Readability.** *All information presented through the interface should be readable by young and old.* The interface designer should emphasize readable type styles, font sizes, and color background choices that enhance contrast.
- **Track state.** *When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.*
- **Visible navigation.** *A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them"*

Nielsen and Wagner suggest a few pragmatic interface design guidelines that provide a nice complement to the principles suggested earlier in this section:

• Reading speed on a computer monitor is approximately 25 percent slower than reading speed for hardcopy. Therefore, do not force the user to read voluminous amounts of text, particularly when the text explains the operation of the WebApp or assists in navigation.

• Avoid "under construction" signs—an unnecessary link is sure to disappoint.

• Users prefer not to scroll. Important information should be placed within the dimensions of a typical browser window.

• Navigation menus and head bars should be designed consistently and should be available on all pages that are available to the user. The design should not rely on browser functions to assist in navigation.

• Aesthetics should never supersede functionality. For example, a simple button might be a better navigation option than an aesthetically pleasing, but vague image or icon whose intent is unclear.
• Navigation options should be obvious, even to the casual user. The user should not have to search the screen to determine how to link to other content or services.

4.5.2 Interface Design Workflow for WebApps

The following tasks represent a rudimentary workflow for WebApp interface design:
1. Review information contained in the requirements model and refine as required.
2. Develop a rough sketch of the WebApp interface layout.
3. Map user objectives into specific interface actions.
4. Define a set of user tasks that are associated with each action.
5. Storyboard screen images for each interface action.
6. Refine interface layout and storyboards using input from aesthetic design.
7. Identify user interface objects that are required to implement the interface.
8. Develop a procedural representation of the user's interaction with the interface.
9. Develop a behavioral representation of the interface.
10. Describe the interface layout for each state.
11. Refine and review the interface design model.

****

4.6 DESIGN EVALUATION

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.

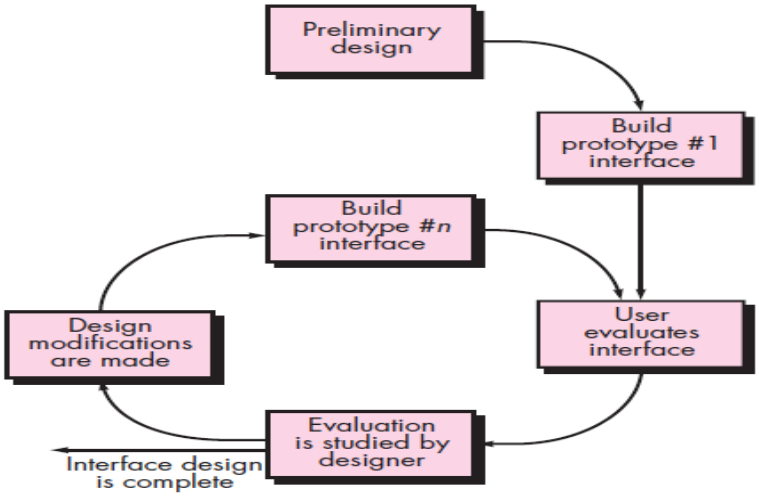


Fig : The interface design evaluation cycle

The user interface evaluation cycle takes the form shown in above figure. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used, you can extract information from these data. Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.

If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

1. The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.

2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.

3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.

4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.
Once the first prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be: (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, (4) Likert scales (e.g., strongly agree, somewhat agree), (5) percentage (subjective) response, or (6) open-ended.
****

4.7 WEBAPP DESIGN QUALITY

Design is the engineering activity that leads to a high-quality product. Olsina and his colleagues have prepared a "quality requirement tree" that identifies a set of technical attributes : **usability, functionality, reliability, efficiency, and maintainability,** that lead to high-quality.

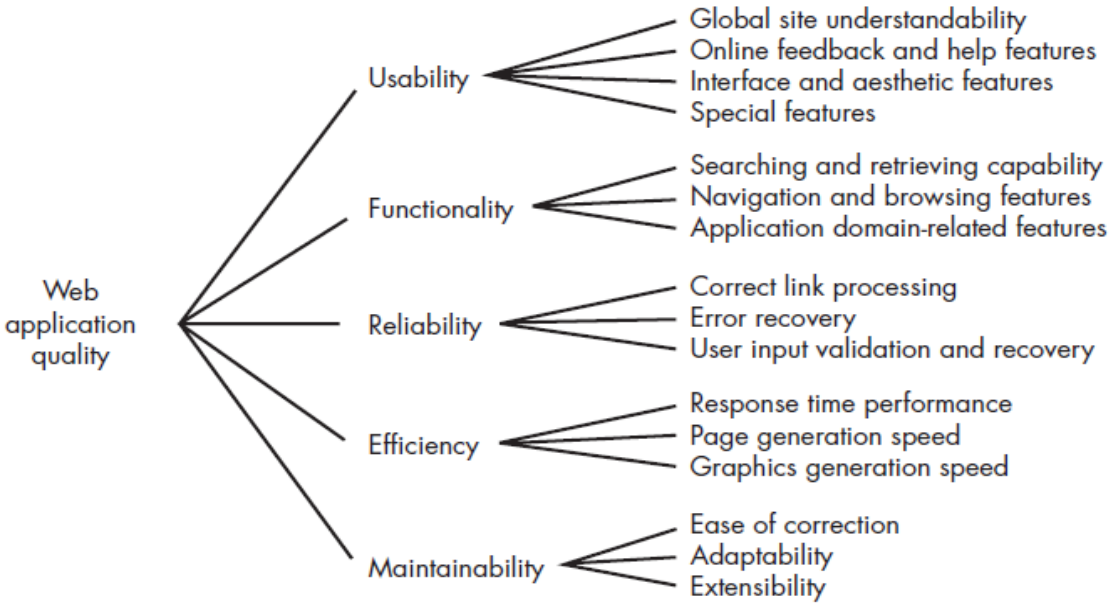


Fig : Quality requirements tree.

Offutt extends the **five** major quality attributes noted in above figure by adding the following attributes:

- **Security.** WebApps have become heavily integrated with critical corporate and government databases. E-commerce applications extract and then store sensitive customer information.
- **Availability.** Even the best WebApp will not meet users' needs if it is unavailable. In a technical sense, availability is the measure of the percentage of time that a WebApp is available for use. The typical end user expects WebApps to be available 24/7/365.
- **Scalability.** Can the WebApp and its server environment be scaled to handle 100, 1000, 10,000, or 100,000 users? Will the WebApp and the systems with which it is interfaced handle significant variation in volume or will responsiveness drop dramatically? It is not enough to build a WebApp that is successful.
- **Time-to-market.** Although time-to-market is not a true quality attribute in the technical sense, it is a measure of quality from a business point of view. The first WebApp to address a specific market segment often captures a disproportionate number of end users.

Tillman suggests a useful set of criteria for assessing the quality of content:
• Can the scope and depth of content be easily determined to ensure that it meets the user's needs?
• Can the background and authority of the content's authors be easily identified?
• Is it possible to determine the currency of the content, the last update, and what was updated?
• Are the content and its location stable ?
In addition to these content-related questions, the following might be added:
• Is content credible?
• Is content unique? That is, does the WebApp provide some unique benefit to those who use it?
• Is content valuable to the targeted user community?
• Is content well organized? Indexed? Easily accessible?

4.8 DESIGN GOAL

Jean Kaiser suggests a set of design goals that are applicable to virtually every WebApp regardless of application domain, size, or complexity:

- **Simplicity.** Although it may seem old-fashioned, the aphorism "all things in moderation" applies to WebApps. There is a tendency among some designers to provide the end user with "too much"—exhaustive content, extreme visuals, intrusive animation, enormous Web pages, the list is long. Better to strive for moderation and simplicity. Content should be informative but succinct and should use a delivery mode (e.g., text, graphics, video, audio) that is appropriate to the information that is being delivered.
- **Consistency.** This design goal applies to virtually every element of the design model. Content should be constructed consistently should present a consistent look across all parts of the WebApp.
- **Identity.** The aesthetic, interface, and navigational design of a WebApp must be consistent with the application domain for which it is to be built.
- **Robustness.** Based on the identity that has been established, a WebApp often makes an implicit "promise" to a user. The user expects robust content and functions that are relevant to the user's needs. If these elements are missing or insufficient, it is likely that the WebApp will fail.
- **Navigability.** I have already noted that navigation should be simple and consistent. It should also be designed in a manner that is intuitive and predictable.
- **Visual Appeal.** Of all software categories, Web applications are unquestionably the most visual, the most dynamic, and the most unapologetically aesthetic. Beauty (visual appeal) is undoubtedly in the eye of the beholder, but many design characteristics (e.g., the look and feel of content; interface layout; color coordination; the balance of text, graphics, and other media; navigation mechanisms) do contribute to visual appeal.
- **Compatibility.** A WebApp will be used in a variety of environments (e.g., different hardware, Internet connection types, operating systems, browsers) and must be designed to be compatible with each.

****

4.9 A DESIGIN PYRAMID FOR WEBAPPS

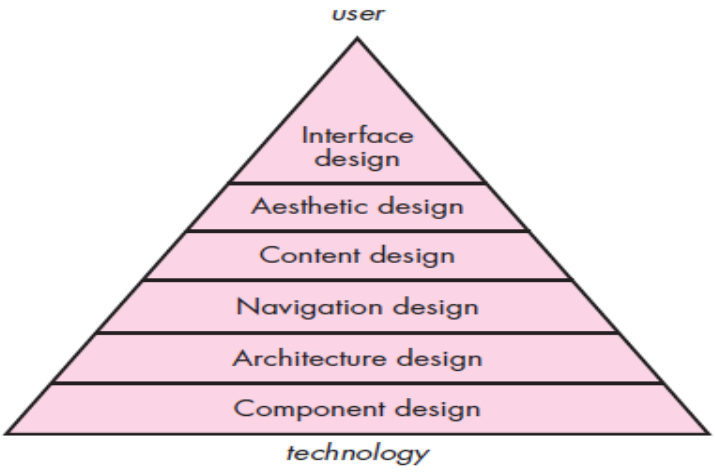


Fig : A design pyramid for WebApps

` The creation of an effective design will typically require a diverse set of skills. Sometimes, for small projects, a single developer may need to be multi-skilled. For larger projects, it may be advisable and/or feasible to draw on the expertise of specialists: Web engineers, graphic designers, content developers, programmers, database specialists, information architects, network engineers, security experts, and testers. Drawing on these diverse skills allows the creation of a model that can be assessed for quality and improved *before* content and code are generated, tests are conducted, and end-users become involved in large numbers. If analysis is where *WebApp quality is established,* then design is where the *quality is truly embedded.*

The appropriate mix of design skills will vary depending upon the nature of the WebApp. The above figure depicts a design pyramid for WebApps. Each level of the pyramid represents a design action that is described in the sections that follow.

4.10 WEBAPP INTERFACE DESIGN

One of the challenges of interface design for WebApps is the indeterminate nature of the user's entry point. That is, the user may enter the WebApp at a "home" location (e.g., the home page) or may be linked into some lower level of the WebApp architecture.
The objectives of a WebApp interface are to:
(1) establish a consistent window into the content and functionality provided by the interface,
(2) guide the user through a series of interactions with the WebApp, and
(3) organize the navigation options and content available to the user.

To achieve a consistent interface, you should first use aesthetic design to establish a coherent "look." This encompasses many characteristics, but must emphasize the layout and form of navigation mechanisms.
To implement navigation options, you can select from one of a number of interaction mechanisms:
• ***Navigation menus***—keyword menus (organized vertically or horizontally) that list key content and/or functionality. These menus may be implemented so that the user can choose from a hierarchy of subtopics that is displayed when the primary menu option is selected.
• ***Graphic icons***—button, switches, and similar graphical images that enable the user to select some property or specify a decision.
• ***Graphic images***—some graphical representation that is selectable by the user and implements a link to a content object or WebApp functionality.
****

4.11 AESTHEIC DESIGN

Aesthetic design, also called ***graphic design***, is an artistic endeavor that complements the technical aspects of WebApp design. Without it, a WebApp may be functional, but unappealing. With it, a WebApp draws its users into a world that embraces them on a visceral, as well as an intellectual level. **Aesthetic is "beauty exists in the eye of the beholder**."

4.11.1 Layout Issues

Every Web page has a limited amount of "real estate" that can be used to support nonfunctional aesthetics, navigation features, informational content, and user-directed functionality. The development of this real estate is planned during aesthetic design. Like all aesthetic issues, there are no absolute rules when screen layout is designed. However, a number of general layout guidelines are worth considering:
Don't be afraid of white space. It is inadvisable to pack every square inch of a Web page with information. The resulting clutter makes it difficult for the user to identify needed information or features and create visual chaos that is not pleasing to the eye.
Emphasize content. After all, that's the reason the user is there. Nielsen suggests that the typical Web page should be 80 percent content with the remaining real estate dedicated to navigation and other features.
Organize layout elements from top-left to bottom-right. The vast majority of users will scan a Web page in much the same way as they scan the page of a book—top-left to bottom-right. If layout elements have specific priorities, high-priority elements should be placed in the upper-left portion of the page real estate.
Group navigation, content, and function geographically within the page. Humans look for patterns in virtually all things. If there are no discernable patterns within a Web page, user frustration is likely to increase.
Don't extend your real estate with the scrolling bar. Although scrolling is often necessary, most studies indicate that users would prefer not to scroll. It is better to reduce page content or to present necessary content on multiple pages.
Consider resolution and browser window size when designing layout. Rather than defining fixed sizes within a layout, the design should specify all layout items as a percentage of available space.

4.11.2 Graphic Design Issues

Graphic design considers every aspect of the look and feel of a WebApp. The graphic design process begins with layout and proceeds into a consideration of global color schemes; text types, sizes, and styles; the use of supplementary media (e.g., audio, video, animation); and all other aesthetic elements of an application.
****

4.12 CONTENT DESIGN

Content design focuses on **two** different design tasks, each addressed by individuals with different skill sets. **First**, a design representation for content objects and the mechanisms required to establish their relationship to one another is developed. In addition, the information within a specific content object is created. The **latter task** may be conducted by copywriters, graphic designers, and others who generate the content to be used within a WebApp.

4.12.1 Content Objects

The relationship between content objects defined as part of a requirements model for the WebApp and design objects representing content is analogous to the relationship between analysis classes and design components.
In the context of WebApp design, a content object is more closely aligned with a data object for traditional software. A content object has attributes that include content-specific information and implementation-specific attributes that are specified as part of design.

4.12.2 Content Design Issues

Once all content objects are modeled, the information that each object is to deliver must be authored and then formatted to best meet the customer's needs. Content authoring is the job of specialists in the relevant area who design the content object by providing an outline of information to be delivered and an indication of the types of generic content objects (e.g., descriptive text, graphic images, photographs) that will be used to deliver the information. Aesthetic design may also be applied to represent the proper look and feel for the content.
As content objects are designed, they are "chunked" to form WebApp pages. The number of content objects incorporated into a single page is a function of user needs, constraints imposed by download speed of the Internet connection, and restrictions imposed by the amount of scrolling that the user will tolerate.

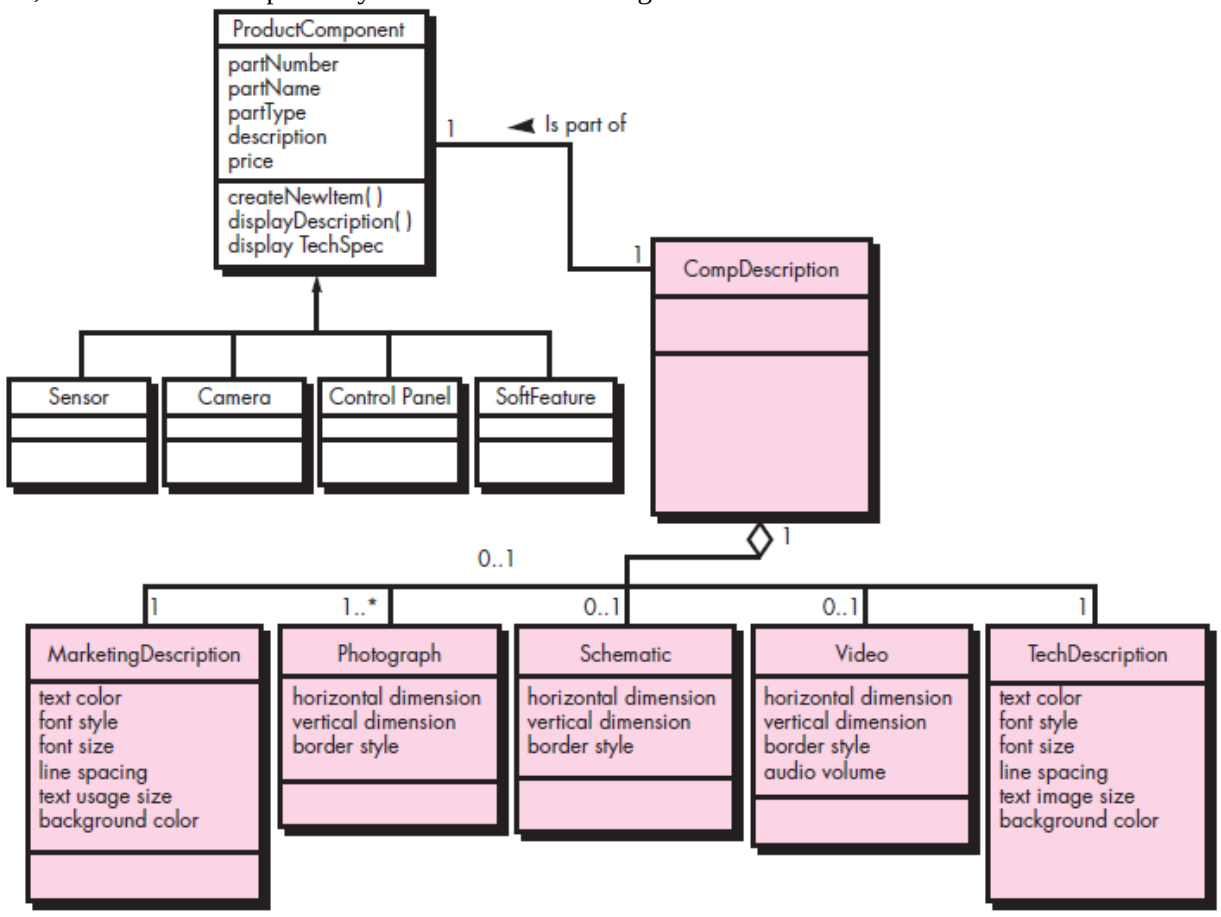


Fig : Design representation of content objects

4.13 ARCHITECTURE DESIGN

Architecture design is tied to the goals established for a WebApp, the content to be presented, the users who will visit, and the navigation philosophy that has been established. As an architectural designer, you must identify **content architecture** and **WebApp architecture**.

Content architecture focuses on the manner in which content objects are structured for presentation and navigation. ***WebApp architecture*** addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.

4.13.1 Content Architecture

The design of content architecture focuses on the definition of the overall hypermedia structure of the WebApp. Although custom architectures are sometimes created, you always have the option of choosing from **four** different content structures

Linear structures : The following figures are encountered when a predictable sequence of interactions (with some variation or diversion) is common. The sequence of content presentation is predefined and generally linear. Another example might be a product order entry sequence in which specific information must be specified in a specific order. As content and processing become more complex, the purely linear flow shown on the left of the figure gives way to more sophisticated linear structures in which alternative content may be invoked or a diversion to acquire complementary content (structure shown on the right side of figure) occurs.

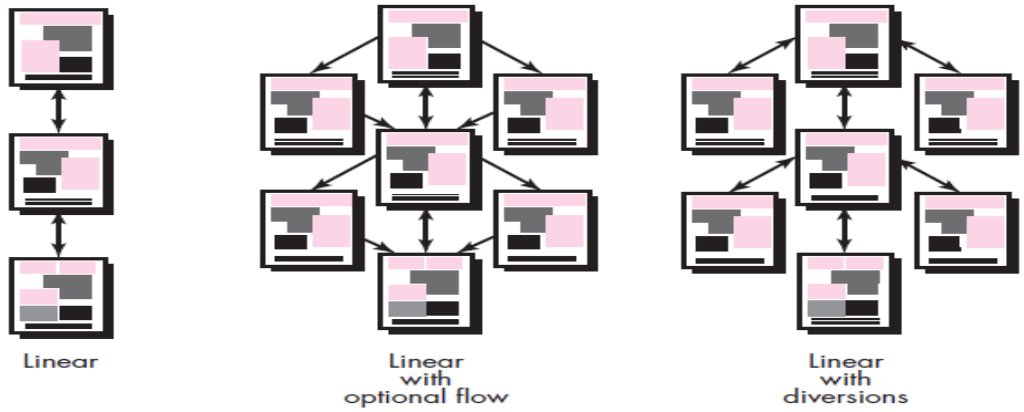


Fig : Linear structures

Grid structures The following figure is an architectural option that you can apply when WebApp content can be organized categorically in two (or more) dimensions. For example, consider a situation in which an e-commerce site sells golf clubs. The horizontal dimension of the grid represents the type of club to be sold (e.g., woods, irons, wedges, putters). The vertical dimension represents the offerings provided by various golf club manufacturers. Hence, a user might navigate the grid horizontally to find the putters column and then vertically to examine the offerings provided by those manufacturers that sell putters. This WebApp architecture is useful only when highly regular content is encountered.

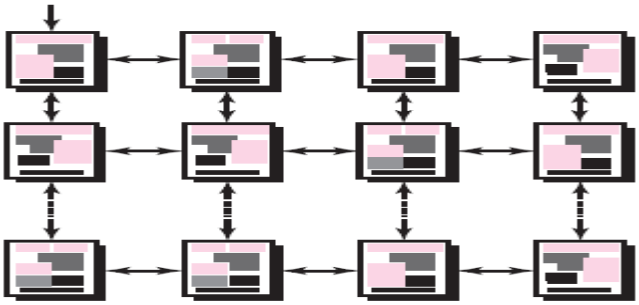


Fig : *Grid structures*

Hierarchical structures The following figure is undoubtedly the most common WebApp architecture. A WebApp hierarchical structure can be designed in a manner that enables (via hypertext branching) flow of control horizontally across vertical branches of the structure. Hence, content presented on **the far left-hand** branch of the hierarchy can have hypertext links that lead directly to content that exists in the **middle or right-hand** branch of the structure. It should be noted, however, that although such branching allows rapid navigation across WebApp content, it can lead to confusion on the part of the user.

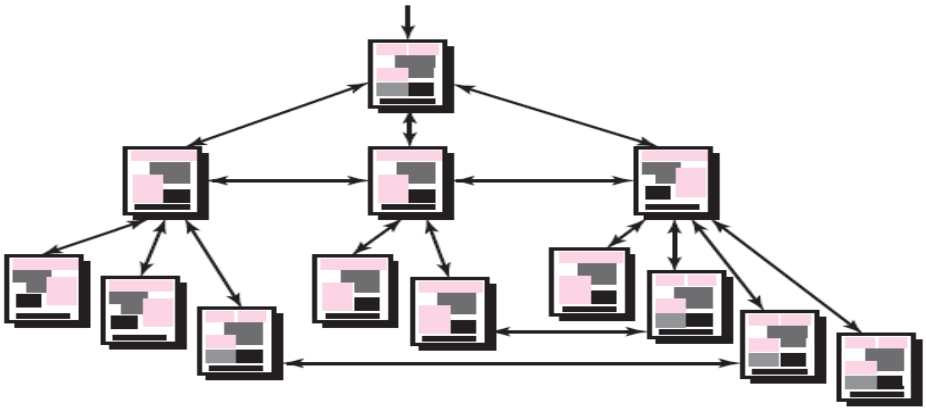


Fig : *Hierarchical structures*

A *networked* or *"pure web" structure* the following figure is similar in many ways to the architecture that evolves for object-oriented systems. Architectural components (in this case, Web pages) are designed so that they may pass control (via hypertext links) to virtually every other component in the system. This approach allows considerable navigation flexibility, but at the same time, can be confusing to a user.

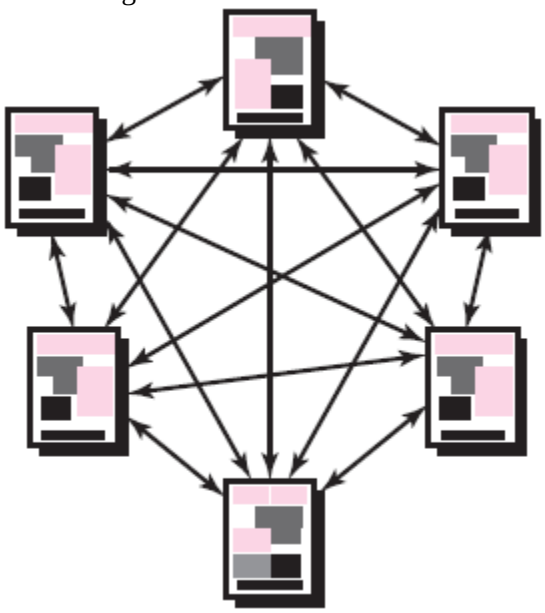


Fig : A *networked* or *"pure web" structure*

The overall architecture of a WebApp may be hierarchical, but part of the structure may exhibit linear characteristics, while another part of the architecture may be networked. Your goal as an architectural designer is to match the WebApp structure to the content to be presented and the processing to be conducted.

4.13.2 WebApp Architecture

WebApp architecture describes an infrastructure that enables a Web-based system or application to achieve its business objectives. Jacyntho and his colleagues describe the basic characteristics of this infrastructure in the following manner: *Applications should be built using layers in which different concerns are taken into account; in particular, application data should be separated from the page's contents and these contents, in turn, should be clearly separated from the interface look-and-feel (pages).*

The authors suggest a **three-layer** design architecture that decouples interface from navigation and from application behavior. They argue that keeping interface, application, and navigation separate simplifies implementation and enhances reuse.

The ***Model-View-Controller*** **(MVC)** architecture is one of a number of suggested WebApp infrastructure models that decouple the user interface from the WebApp functionality and informational content. The MVC architecture decouples the user interface from WebApp functionality and information content.

The ***model*** (sometimes referred to as the "model object") contains all application-specific content and processing logic, including all content objects, access to external data/information sources, and all processing functionality that is application specific.

The ***view*** contains all interface specific functions and enables the presentation of content and processing logic, including all content objects, access to external data/information sources, and all processing functionality required by the end user.

The ***controller*** manages access to the model and the view and coordinates the flow of data between them. In a WebApp, "the view is updated by the controller with data from the model based on user input". A schematic representation of the MVC architecture is shown in following figure.

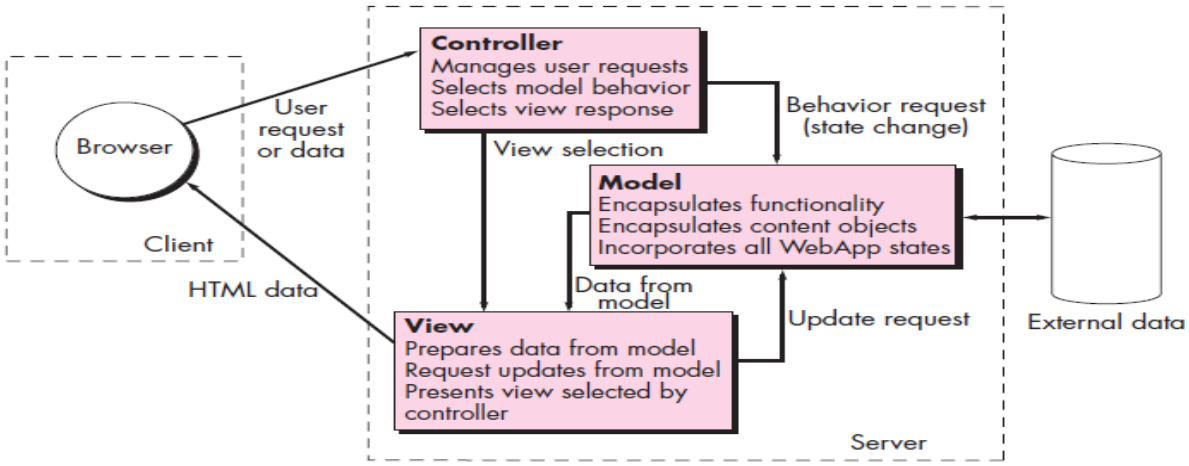


Fig : The MVC Architecture

Referring to the figure, user requests or data are handled by the controller. The controller also selects the view object that is applicable based on the user request. Once the type of request is determined, a behavior request is transmitted to the model, which implements the functionality or retrieves the content required to accommodate the request. The model object can access data stored in a corporate database, as part of a local data store, or as a collection of independent files. The data developed by the model must be formatted and organized by the appropriate view object and then transmitted from the application server back to the client-based browser for display on the customer's machine.

****

4.14 NAVIGATION DESIGN

4.14.1 Navigation Semantics

Like many WebApp design actions, navigation design begins with a consideration of the user hierarchy and related use cases developed for each category of user (actor). Each actor may use the WebApp somewhat differently and therefore have different navigation requirements. In addition, the use cases developed for each actor will define a set of classes that encompass one or more content objects or WebApp functions. As each user interacts with the WebApp, she encounters a series of ***navigation semantic units*** **(NSUs). NSU** is "a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements".

An NSU is composed of a set of navigation elements called ***ways of navigating*****(WoN**). A WoN represents the best navigation pathway to achieve a navigational goal for a specific type of user. Each WoN is organized as a set of ***navigational nodes*** **(NN)** that are connected by navigational links. The overall navigation structure for a WebApp may be organized as a hierarchy of NSUs.

4.14.2 Navigation Syntax

As design proceeds, your next task is to define the mechanics of navigation. A number of options are available as you develop an approach for implementing each NSU:

• ***Individual navigation link***—includes text-based links, icons, buttons and switches, and graphical metaphors. You must choose navigation links that are appropriate for the content and consistent with the heuristics that lead to high-quality interface design.

• ***Horizontal navigation bar***—lists major content or functional categories in a bar containing appropriate links. In general, between four and seven categories are listed.

• ***Vertical navigation column***—(1) lists major content or functional categories, or (2) lists virtually all major content objects within the WebApp. If you choose the second option, such navigation columns can "expand" to present content objects as part of a hierarchy

• ***Tabs***—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.

• ***Site maps***—provide an all-inclusive table of contents for navigation to all content objects and functionality contained within the WebApp.

****

4.15 COMPONENT-LEVEL DESIGN

Modern WebApps 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 are appropriate for the WebApp's business domain, (3) provide sophisticated database query and access, and (4) establish data interfaces with external corporate systems. To achieve these capabilities, you must design and construct program components that are identical in form to software components for traditional software.

****

4.16 OBJECT-ORIENTED HYPERMEDIA DESIGN METHOD (OOHMD)

Daniel Schwabe and his colleagues originally proposed the ***Object-Oriented Hypermedia Design Method*** **(OOHDM)**, which is composed of **four** different design activities: **conceptual design, navigational design, abstract interface design, and implementation.** A summary of these design activities is shown in following table

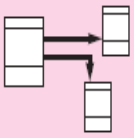
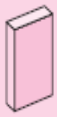
	 Conceptual design	 Navigational design	 Abstract interface design	 Implementation
Work products	Classes, subsystems, relationships, attributes	Nodes links, access structures, navigational contexts, navigational transformations	Abstract interface objects, responses to external events, transformations	Executable WebApp
Design mechanisms	Classification, composition, aggregation, generalization specialization	Mapping between conceptual and navigation objects	Mapping between navigation and perceptible objects	Resource provided by target environment
Design concerns	Modeling semantics of the application domain	Takes into account user profile and task. Emphasis on cognitive aspects.	Modeling perceptible objects, implementing chosen metaphors. Describe interface for navigational objects.	Correctness; application performance; completeness

Table : Summary of the OOHDM method

4.16.1 Conceptual Design for OOHDM

OOHDM ***conceptual design*** creates a representation of the subsystems, classes, and relationships that define the application domain for the WebApp. UML may be used to create appropriate class diagrams, aggregations, and composite class representations, collaboration diagrams, and other information that describes the application domain.

4.16.2 Navigational Design for OOHDM

Navigational design identifies a set of "**objects**" that are derived from the classes defined in conceptual design. A series of "**navigational classes**" or "**nodes**" are defined to encapsulate these objects. UML may be used to create appropriate use cases, state charts, and sequence diagrams. **OOHDM** uses a predefined set of navigation classes such as nodes, links, anchors, and access structures. Access structures are more elaborate and include mechanisms such as a WebApp index, a site map, or a guided tour.

Once navigation classes are defined, OOHDM "structures the navigation space by grouping navigation objects into sets called contexts". A ***context*** includes a description of the local navigation structure, restriction imposed on the access of content objects, and methods (operations) required to effect access of content objects.

4.16.3 Abstract Interface Design and Implementation

The ***abstract interface design*** action specifies the interface objects that the user sees as WebApp interaction occurs. A formal model of interface objects, called an ***abstract data view*** (ADV), is used to represent the relationship between interface objects and navigation objects, and the behavioral characteristics of interface objects. The ADV model defines a "**static layout**" that represents the interface metaphor and includes a representation of navigation objects within the interface and the specification of the interface objects (e.g., menus, buttons, icons) that assist in navigation and interaction.

In addition, the ADV model contains a behavioral component (similar to the UML state diagram) that indicates how external events "trigger navigation and which interface transformations occur when the user interacts with the application".

The OOHDM *implementation* activity represents a design iteration that is specific to the environment in which the WebApp will operate. Classes, navigation, and the interface are each characterized in a manner that can be constructed for the client-server environment, operating systems, support software, programming languages, and other environmental characteristics that are relevant to the problem

UNIT- V

Software Testing Strategies: A strategic Approach to Software Testing, Strategic Issues, Test Strategies for Conventional Software, Test Strategies for Object-Oriented Software, Test Strategies for WebApps, Validation Testing, System Testing, The Art of Debugging.
Testing Conventional Applications: Software Testing Fundamentals, Internal and External Views of Testing, White-Box Testing, basic Path testing, Control Structure Testing, Black-Box Testing, Model-based Testing, Testing for Specialized Environments, Architectures and Applications, Patterns for Software Testing.
Testing Object-Oriented Applications: Broadening the View of Testing, Testing with OOA and OOD Models, Object-Oriented Testing Strategies, Object-Oriented Testing Methods, Testing Methods Applicable at the Class level, Interclass Test-Case Design.

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 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.

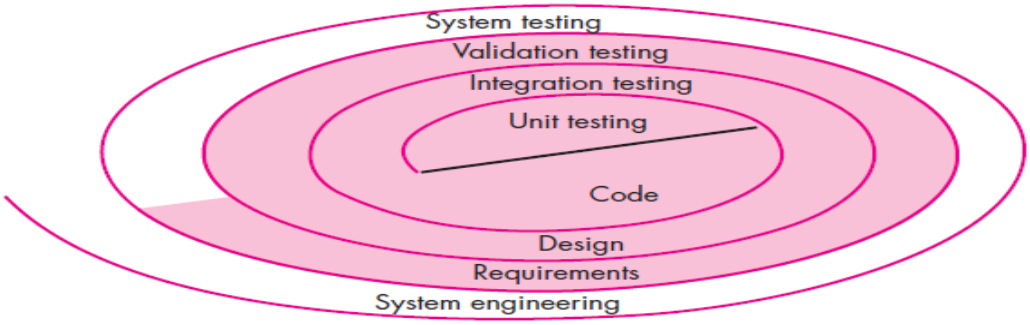


Fig : Testing Strategy

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.

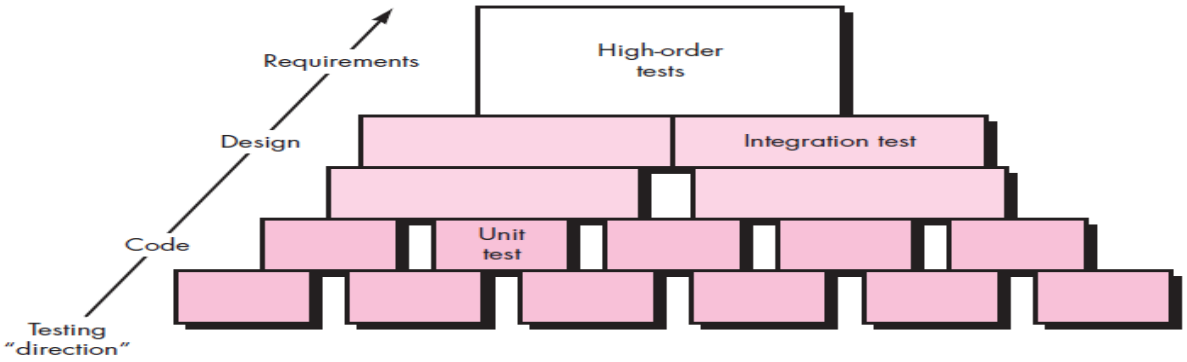


Fig : Software testing steps

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.

5.1.4 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?"

****

5.2 TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

Tom Gilb argues that a software testing strategy will succeed when software testers:

- ***Specify product requirements in a quantifiable manner long before testing commences***.

Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability.. These should be specified in a way that is measurable so that testing results are unambiguous.

- ***State testing objectives explicitly***. The specific objectives of testing should be stated in measurable terms.
- ***Understand the users of the software and develop a profile for each user category***. Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.
- ***Develop a testing plan that emphasizes "rapid cycle testing."*** Gilb recommends that a software team "learn to test in rapid cycles The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.
- ***Build "robust" software that is designed to test itself***. Software should be designed in a manner that uses anti bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
- ***Use effective technical reviews as a filter prior to testing***. Technical reviews can be as effective as testing in uncovering errors.
- ***Conduct technical reviews to assess the test strategy and test cases themselves***. Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
- ***Develop a continuous improvement approach for the testing process***. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

****

5.3 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.

5.3.1 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 operates properly at boundaries established to limit or restrict processing. And finally, all **error-handling paths** are tested.

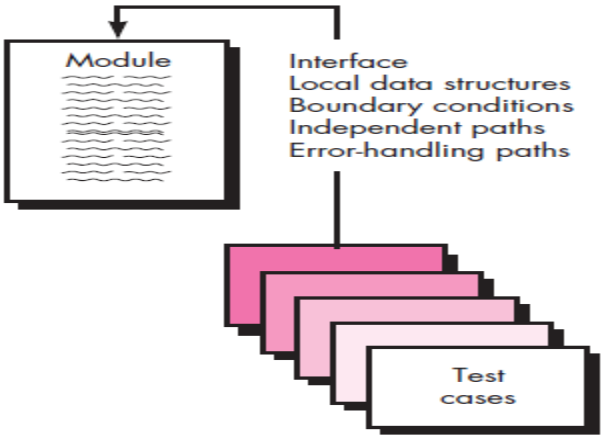


Fig : Unit Test

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.

5.3.2 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.

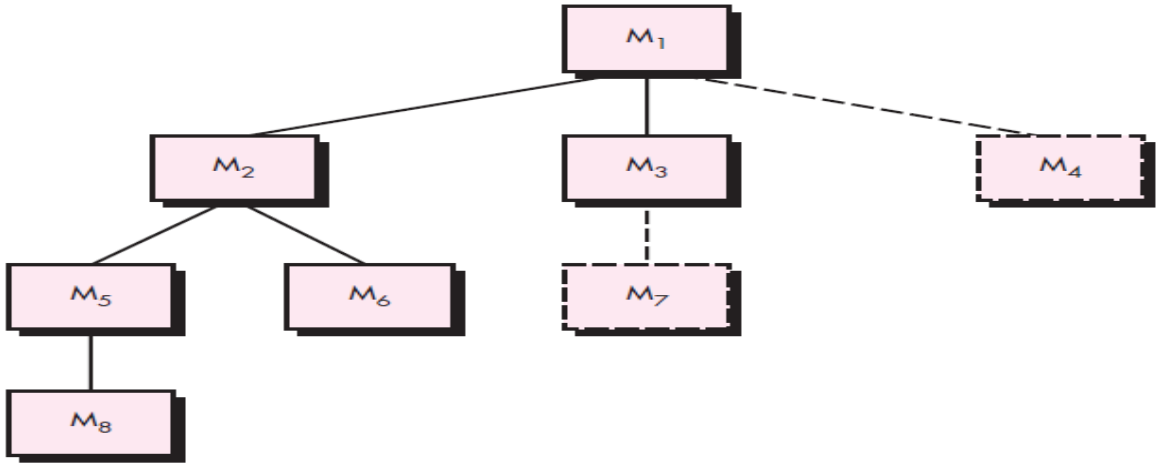


Fig : Top-down integration

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.

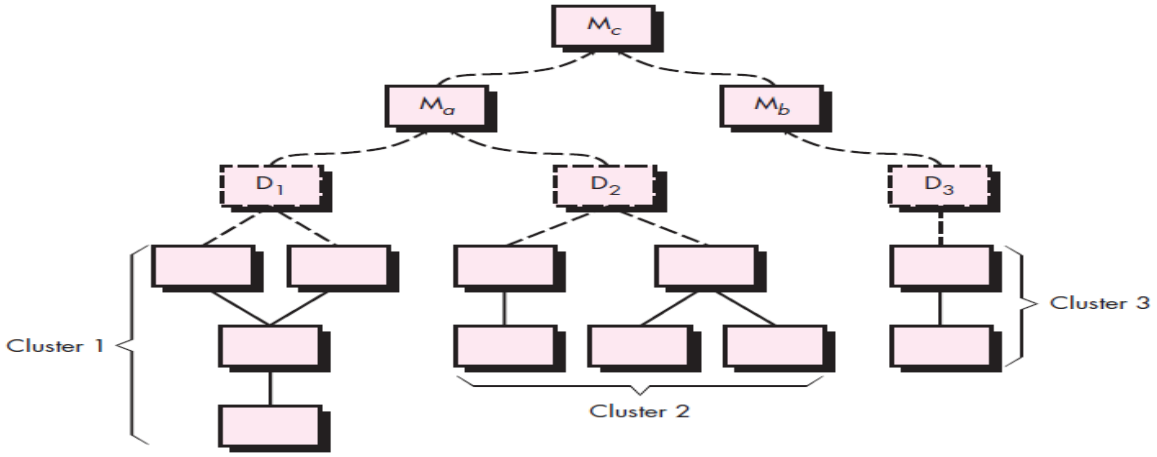


Fig : Bottom-up integration

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

5.4 TEST STRATEGIES FOR OBJECT-ORIENTED SOFTWARE

5.4.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data. An encapsulated class is usually the focus of **unit testing**.
Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

5.4.2 Integration Testing in the OO Context

There are two different strategies for integration testing of OO systems.
The first, ***thread-based testing***, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur.
The second integration approach, ***use-based testing***, begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called ***dependent classes***, that use the independent classes are tested.
Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations.
****

5.5 TEST STRATEGIES FOR WEBAPPS

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:
1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.
****

5.6 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.

5.6.1 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.

5.6.2 Configuration Review

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

5.6.3 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.
****

5.7 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.

5.7.1 Recovery Testing

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

5.7.2 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.

5.7.3 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.

5.7.4 Performance Testing

Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

5.7.5 Deployment Testing

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

5.8 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.

5.8.1 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.

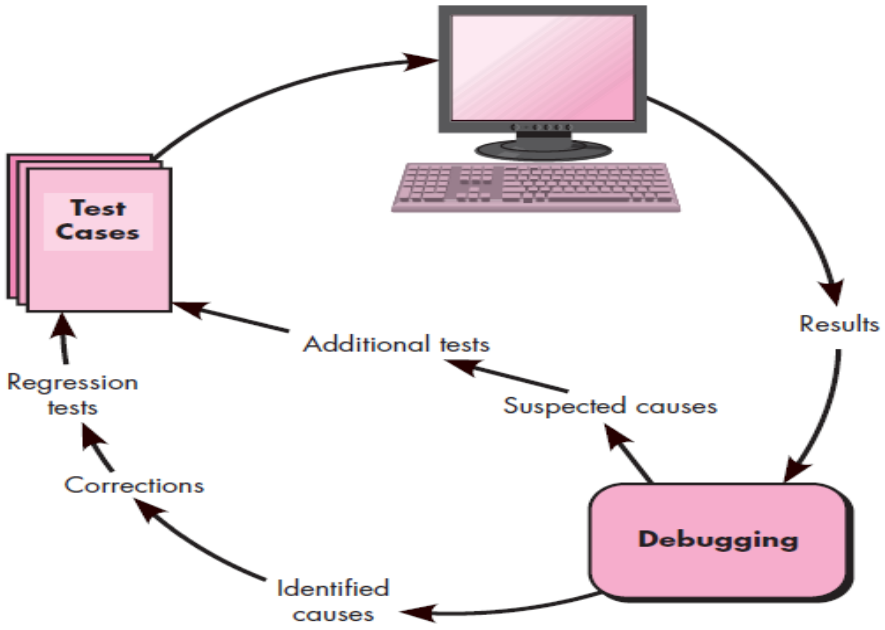


Fig : The Debugging Process

5.8.2 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.

5.8.3 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

5.8.4 Correcting the Error

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. 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.

****

5.9 SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer based system or a product with "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability. James Bach provides the following definition for testability: "*Software testability* is simply how easily can be tested." The following **characteristics** lead to testable software.

Operability. "The better it works, the more efficiently it can be tested."

Observability. "What you see is what you test."

Controllability. "The better we can control the software, the more the testing can be automated and optimized."

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

Simplicity. "The less there is to test, the more quickly we can test it." The program should exhibit *functional simplicity , structural simplicity,* and *code simplicity*

Stability. "The fewer the changes, the fewer the disruptions to testing."

Understandability. "The more information we have, the smarter we will test."

Test Characteristics. Kaner, Falk, and Nguyen suggest the following attributes of a "good" test:

A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed.

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.

A good test should be "best of breed" In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

5.10 INTERNAL AND EXTERNAL VIEWS OF TESTING

Any engineered product can be tested in one of **two** ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. (2) Knowing the internal workings of a product.
The first test approach takes an **external view** and is called **black-box testing.** The second requires an **internal view** and is termed **white-box testing**.
Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.
White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.
****

5.11 WHITE-BOX TESTING

White-box testing, sometimes called ***glass-box testing***, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at least once,
(2) exercise all logical decisions on their true and false sides,
(3) execute all loops at their boundaries and within their operational bounds, and
(4) exercise internal data structures to ensure their validity.

****

5.12 BASIC PATH TESTING

Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test-case designer to derive a **logical complexity measure** of a procedural design and use this measure as a guide for defining a basis **set of execution paths**. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program **at least one time** during testing.

5.12.1 Flow Graph Notation

A simple notation for the representation of control flow, called a *flow graph* (or *program graph*). The flow graph depicts logical control flow using the notation illustrated in following figure.

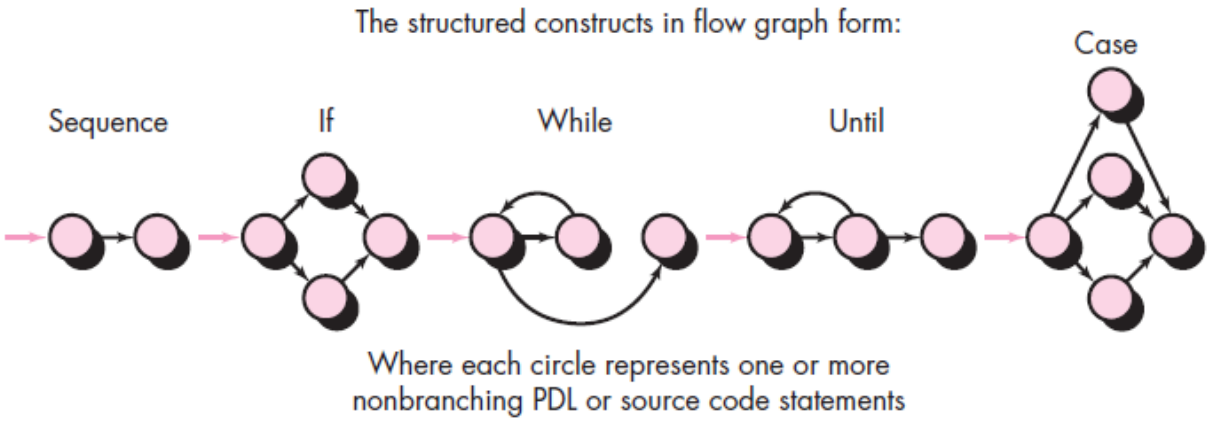


Fig : Flow Graph Notation

To illustrate the use of a flow graph, consider the procedural design representation in following figure (a). Here, a flowchart is used to depict program control structure. Figure (b) maps the flowchart into a corresponding flow graph.
Referring to figure (b), each circle, called a ***flow graph node***, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called ***edges*** **or** ***links***, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements. Areas bounded by edges and nodes are called ***regions***. When counting regions, we include the area outside the graph as a region Each node that contains a condition is called a ***predicate node*** and is characterized by two or more edges emanating from it

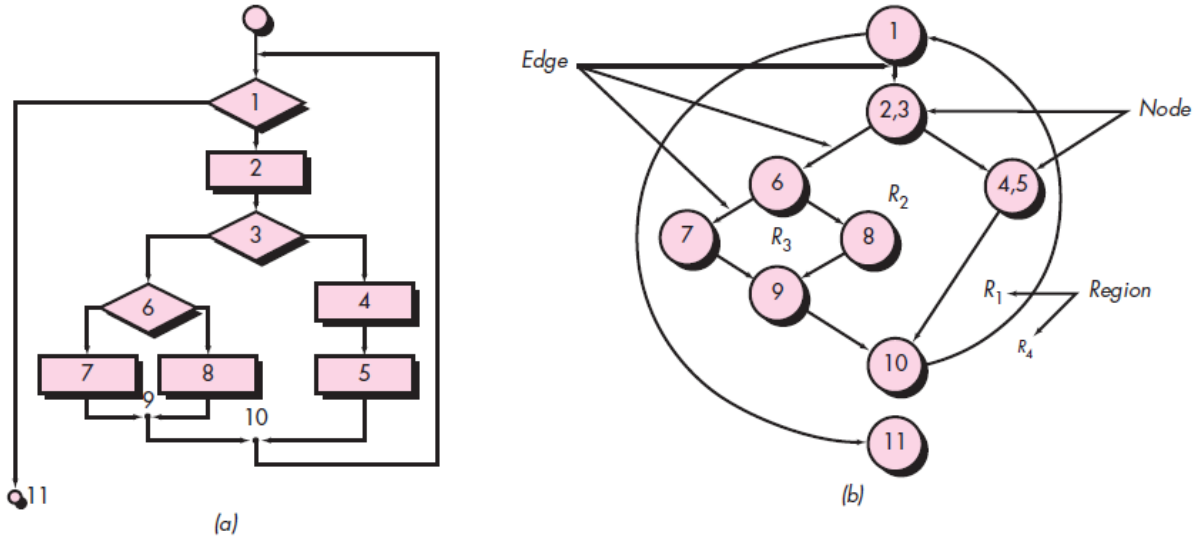


Fig : (a) Flowchart and (b) flow graph

5.12.2 Independent Program Paths

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

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

Note that each new path introduces a new edge. The path
1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

How do you know how many paths to look for? The computation of **cyclomatic complexity** provides the answer. ***Cyclomatic complexity*** is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of **three** ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.

2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as

$V(G) = E - N + 2$;

where E is the number of flow graph edges and N is the number of flow graph nodes.

3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as

$V(G) = P + 1$

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

Referring once more to the flow graph in figure (b), the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has **four** regions.
2. $V(G)$ = 11 edges - 9 nodes + = 4.
3. $V(G)$ = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in figure (b) is 4.

5.12.3 Deriving Test Cases

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

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

2. Determine the cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

5.12.4 Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. A data structure, called a ***graph matrix***, can be quite useful for developing a software tool that assists in basis path testing.

A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix is shown in following figure..

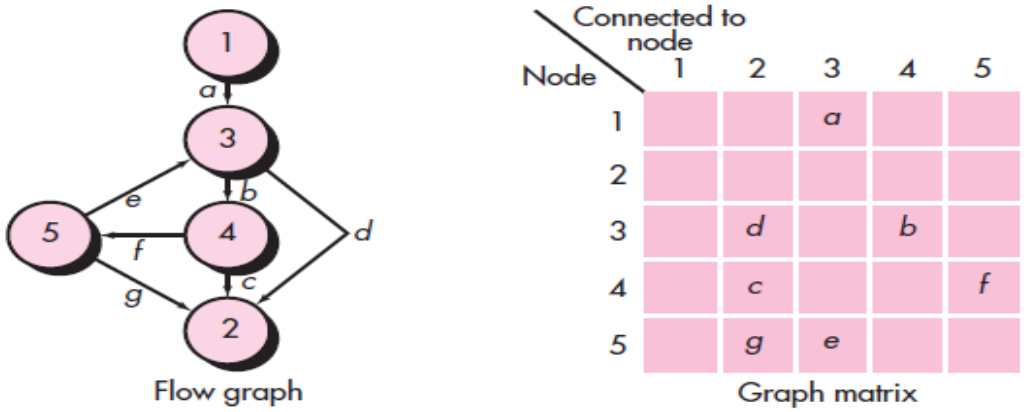


Fig : Graph Matrix

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*. To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a ***link weight*** to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.

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

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

****

5.13 CONTROL STRUCTURE TESTING

These broaden control structure testing coverage and improve the quality of white-box testing.

5.13.1 Condition Testing

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

***E*1 <relational-operator> *E*2**

where *E*1 and *E*2 are arithmetic expressions and <relational-operator> is one of the following: <,<=, =, ≠,> or >=. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors,

Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

5.13.2 Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables.

5.13.3 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software.

Loop testing is a **white-box testing technique** that focuses exclusively on the validity of loop constructs. **Four** different classes of loops can be defined: **simple loops, concatenated loops, nested loops, and unstructured loops** (shown in figure).

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

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

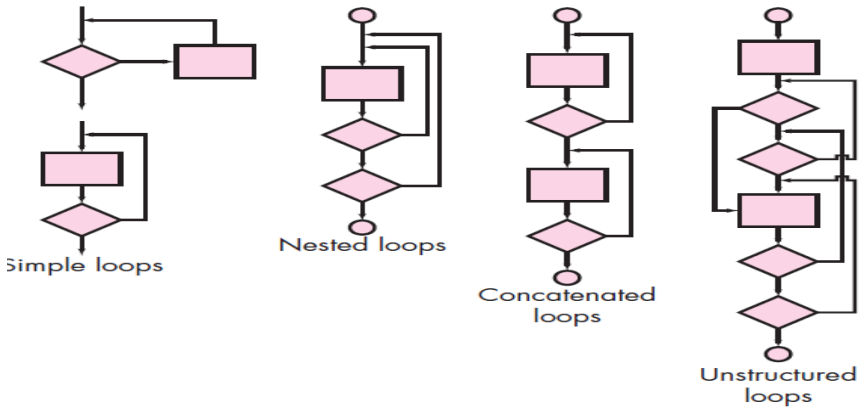


Fig : Classes of Loops

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

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

Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

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

****

5.14 BLACK-BOX TESTING

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

Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, 4) behavior or performance errors, and (5) initialization and termination errors.

Tests are designed to answer the following questions:

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

5.14.1 Graph-Based Testing Methods

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

To accomplish these steps, you begin by creating a ***graph***, it is a collection of ***nodes*** that represent objects, ***links*** that represent the relationships between objects, ***node weights*** that describe the properties of a node, and ***link weights*** that describe some characteristic of a link.

The symbolic representation of a graph is shown in following figure. Nodes are represented as circles connected by links that take a number of different forms.

A ***directed link*** (represented by an arrow) indicates that a relationship moves in only one direction. A ***bidirectional link***, also called a ***symmetric link***, implies that the relationship applies in both directions. ***Parallel links*** are used when a number of different relationships are established between graph nodes.

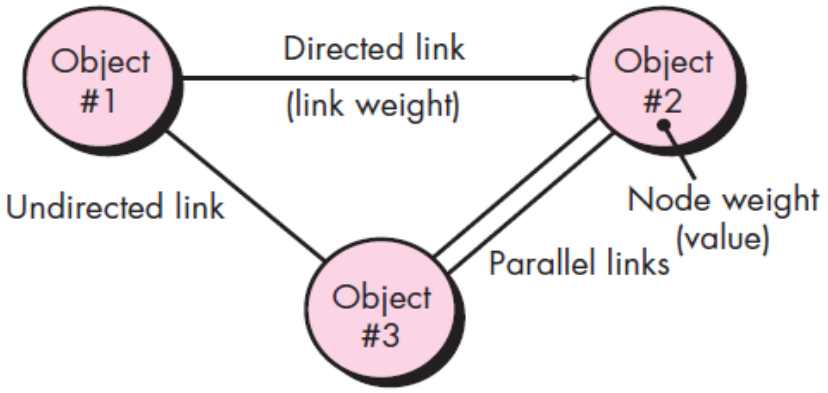


Fig : Graph Notation

Beizer describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction, and the links represent the logical connection between steps .

Finite state modeling. The nodes represent different user-observable states of the software, and the links represent the transitions that occur to move from state to state. The state diagram can be used to assist in creating graphs of this type.
Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another.
Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

5.14.2 Equivalence Partitioning

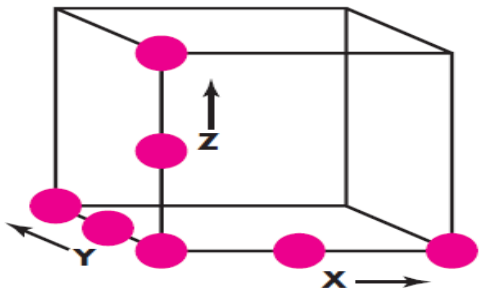
Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. Test-case design for equivalence partitioning is based on an evaluation of ***equivalence classes*** for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present.
Equivalence classes may be defined according to the following guidelines:
1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

5.14.3 Boundary Value Analysis

A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that ***boundary value analysis*** **(BVA)** has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.
Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.
Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:
1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary. Most software engineers intuitively perform BVA to some degree.

5.14.4 Orthogonal Array Testing

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding ***region faults***—an error category associated with faulty logic within a software component.
Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases.



One input item at a time

****

5.15 MODEL-BASED TESTING

Model-based testing **(MBT)** is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases. In many cases, the model-based testing technique uses UML state diagrams, an element of the behavioral model, as the basis for the design of test cases.

The MBT technique requires **five** steps:

1. Analyze an existing behavioral model for the software or create one. Recall that a *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the 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 UML state diagram for the system and (5) review the behavioral model to verify accuracy and consistency.

2. Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state. The inputs will trigger events that will cause the transition to occur.

3. Review the behavioral model and note the expected outputs as the software makes the transition from state to state. Recall that each state transition is triggered by an event and that as a consequence of the transition, some function is invoked and outputs are created.

4. Execute the test cases. Tests can be executed manually or a test script can be created and executed using a testing tool.

5. Compare actual and expected results and take corrective action as required.

MBT helps to uncover errors in software behavior, and as a consequence, it is extremely useful when testing event-driven applications.

****

5.16 TESTING FOR SPECIALIZED ENVIRONMENTS, ARCHITECTURES AND APPLICATIONS

5.16.1 Testing GUIs

Graphical user interfaces (GUIs) will present you with interesting testing challenges.

Because many modern GUIs have the same look and feel a series of standard tests can be derived. Finite-state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI. Because of the large number of permutations associated with GUI operations, GUI testing should be approached using automated tools.

5.16.2 Testing of Client-Server Architectures

The distributed nature of client-server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized database, and the coordination requirements imposed on the server all combine to make testing of client-server architectures and the software that resides within them considerably more difficult than stand-alone applications.

In general, the testing of client-server software occurs at **three** different levels:

(1) Individual client applications are tested in a "disconnected" mode; the operation of the server and the underlying network are not considered. (2) The client software and associated server applications are tested in concert, but network operations are not explicitly exercised. (3) The complete client-server architecture, including network operation and performance, is tested.

The following testing approaches are commonly encountered for client-server applications:

- **Application function tests.** The application is tested in stand-alone fashion in an attempt to uncover errors in its operation.
- **Server tests.** The coordination and data management functions of the server are tested. Server performance is also considered.
- **Database tests.** The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated, and retrieved. Archiving is also tested.
- **Transaction tests.** A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues.
- **Network communication tests.** These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests.

To accomplish these testing approaches, Musa recommends the development of ***operational profiles*** derived from client-server usage scenarios. An operational profile indicates how different types of users interoperate with the client-server system. That is, the profiles provide a "pattern of usage" that can be applied when tests are designed and executed.

5.16.3 Testing Documentation and Help Facilities

The term *software testing* conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. it is important to note that testing must
also extend to the third element of the software configuration—documentation.
Documentation testing can be approached in **two** phases. The **first** phase, technical review, examines the document for editorial clarity. The **second** phase, live test, uses the documentation in conjunction with the actual program.

5.16.4 Testing for Real-Time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix time. Test data provided when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.
Comprehensive test-case design methods for real-time systems continue to evolve. However, an overall **four-step strategy** can be proposed:
• **Task testing.** The first step in the testing of real-time software is to test each task independently. That is, conventional tests are designed for each task and executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior.
• **Behavioral testing.** Using system models created with automated tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events.
• **Inter task testing.** Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if inter task synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
• **System testing.** Software and hardware are integrated, and a full range of system tests are conducted in an attempt to uncover errors at the software hardware interface. Most real-time systems process interrupts.
Tests are then designed to assess the following system characteristics:
• Are interrupt priorities properly assigned and properly handled?
• Is processing for each interrupt handled correctly?
• Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?
• Does a high volume of interrupts arriving at critical times create problems in function or performance?

****

5.17 PATTERNS FOR SOFTWARE TESTING

The use of patterns as a mechanism for describing solutions to specific design problems. But patterns can also be used to propose solutions to other software engineering situations of software testing. Testing patterns describe common testing problems and solutions that can assist you in dealing with them. Not only do testing patterns provide you with useful guidance as testing activities commence, they also provide three additional benefits described by Marick
1. They [patterns] provide a vocabulary for problem-solvers. "Hey, you know, we should use a Null Object."
2. They focus attention on the forces behind a problem. That allows [test case] designers to better understand when and why a solution applies.
3. They encourage iterative thinking. Each solution creates a new context in which new problems can be solved.

****

5.18 BROADENING THE VIEW OF TESTING

The construction of object-oriented software begins with the creation of requirements (analysis) and design models. Because of the evolutionary nature of the OO software engineering paradigm, these models begin as relatively informal representations of system requirements and evolve into detailed models of classes, class relationships, system design and allocation, and object design. At each stage, the models can be "tested" in an attempt to uncover errors prior to their propagation to the next iteration. It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code levels.

By eliminating the extraneous attribute at this stage, the following problems and unnecessary effort may be avoided during analysis:

1. Special subclasses may have been generated to accommodate the unnecessary attribute or exceptions to it. Work involved in the creation of unnecessary subclasses has been avoided.

2. A misinterpretation of the class definition may lead to incorrect or extraneous class relationships.

3. The behavior of the system or its classes may be improperly characterized to accommodate the extraneous attribute.

If the problem is not uncovered during analysis and propagated further, the following problems could occur during design:

- **a.** Improper allocation of the class to subsystem and/or tasks may occur during system design.
- **b.** Unnecessary design work may be expended to create the procedural design for the operations that address the extraneous attribute.
- **c.** The messaging model will be incorrect

During latter stages of their development, object-oriented analysis (OOA) and design (OOD) models provide substantial information about the structure and behavior of the system. For this reason, these models should be subjected to rigorous review prior to the generation of code. All object-oriented models should be tested for correctness, completeness, and consistency within the context of the model's syntax, semantics, and pragmatics.

****

5.19 TESTING WITH OOA AND OOD MODELS

5.19.1 Correctness of OOA and OOD Models

The notation and syntax used to represent analysis and design models will be tied to the specific analysis and design methods that are chosen for the project. Hence syntactic correctness is judged on proper use of the symbology. During analysis and design, you can assess semantic correctness based on the model's conformance to the real-world problem domain.

5.19.2 Consistency of Object-Oriented Models

The consistency of object-oriented models may be judged by "considering the relationships among entities in the model. An inconsistent analysis or design model has representations in one part that are not correctly reflected in other portions of the model" .

To assess consistency, you should examine each class and its connections to other classes. The class-responsibility-collaboration (CRC) model or an object relationship diagram can be used to facilitate this activity. The object relationship model provides a graphic representation of the connections between classes.

To evaluate the class model the following steps have been recommended

1. Revisit the CRC model and the object-relationship model. Cross-check to ensure that all collaborations implied by the requirements model are properly reflected in the both.
2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
5. Determine whether widely requested responsibilities might be combined into a single responsibility.

****

5.20 OBJECT-ORIENTED TESTING STRATEGIES

5.20.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.

5.20.2 Integration Testing in the OO Context

There are **two** different strategies for integration testing of OO systems. The **first, *thread-based testing***, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually.

Regression testing is applied to ensure that no side effects occur. The **second** integration approach, ***use-based testing***, begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) of server classes.
After the independent classes are tested, the next layer of classes, called ***dependent classes***, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.
Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations.

5.20.3 Validation Testing in an OO Context

Conventional black-box testing methods can be used to drive validation tests. In addition, you may choose to derive test cases from the object behavior model and from an event flow diagram created as part of OOA.
****

5.21 OBJECT-ORIENTED TESTING METHODS

The architecture of object-oriented software results in a series of layered subsystems that encapsulate collaborating classes. Each of these system elements (subsystems and classes) performs functions that help to achieve system requirements.
Test-case design methods for object-oriented software continue to evolve. However, an overall approach to OO test-case design has been suggested by Berard
1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain:
a. A list of specified states for the class that is to be tested
b. A list of messages and operations that will be exercised as a consequence of the test
c. A list of exceptions that may occur as the class is tested
d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
e. Supplementary information that will aid in understanding or implementing the test

5.21.1 The Test-Case Design Implications of OO Concepts

As a class evolves through the requirements and design models, it becomes a target for test-case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing.
If subclasses instantiated from a super class are used within the same problem domain, it is likely that the set of test cases derived for the super class can be used when testing the subclass. However, if the subclass is used in an entirely different context, the super class test cases will have little applicability and a new set of tests must be designed.

5.21.2 Applicability of Conventional Test-Case Design Methods

The white-box testing methods can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested.
Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods.

5.21.3 Fault-Based Testing

The object of ***fault-based testing*** within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, preliminary planning required to perform fault based testing begins with the analysis model. The tester looks for plausible faults. To determine whether these faults exist, test cases are designed to exercise the design or code.

5.21.4 Test Cases and the Class Hierarchy

Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.

5.21.5 Scenario-Based Test Design

Fault-based testing misses **two** main types of errors: (1) incorrect specifications and (2) interactions among subsystems.

Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use cases) that the user has to perform and then applying them and their variants as tests. Scenarios uncover interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test.

5.21.6 Testing Surface Structure and Deep Structure

Surface structure refers to the externally observable structure of an OO program. That is, the structure that is immediately obvious to an end user. Rather than performing functions, the users of many OO systems may be given objects to manipulate in some way.

Deep structure refers to the internal technical details of an OO program, that is, the structure that is understood by examining the design and/or code. Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the design model for OO software.

****

5.22 TESTING METHODS APPLICABLE AT THE CLASS LEVEL

5.22.1 Random Testing for OO Classes

The number of possible permutations for random testing can grow quite large. A strategy similar to orthogonal array testing can be used to improve testing efficiency.

5.22.2 Partition Testing at the Class Level

Partition testing reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning for traditional software. Input and output are categorized and test cases are designed to exercise each category.

State-based partitioning categorizes class operations based on their ability to change the state of the class.

Attribute-based partitioning categorizes class operations based on the attributes that they use.

****

5.23 INTERCLASS TEST-CASE DESIGN

5.23.1 Multiple Class Testing

Kirani and Tsai suggest the following sequence of steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
3. For each operation in the server object, determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

5.23.2 Tests Derived from Behavior Models

The use of the state diagram as a model that represents the dynamic behavior of a class. The state diagram for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class.

\

****