UNIT – 3

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.

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model, the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.
Who does it? Software engineers conduct each of the design tasks. Why is it important? Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers. Design is the place where software quality is established.
What are the steps? Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed. Each of these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.
What is the work product? A design model that encompasses architectural, interface, component level, and deployment representations is the primary work product that is produced during software design.
How do I ensure that I've done it right? The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established.

3.1 DESIGN WITHIN THE 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. Software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

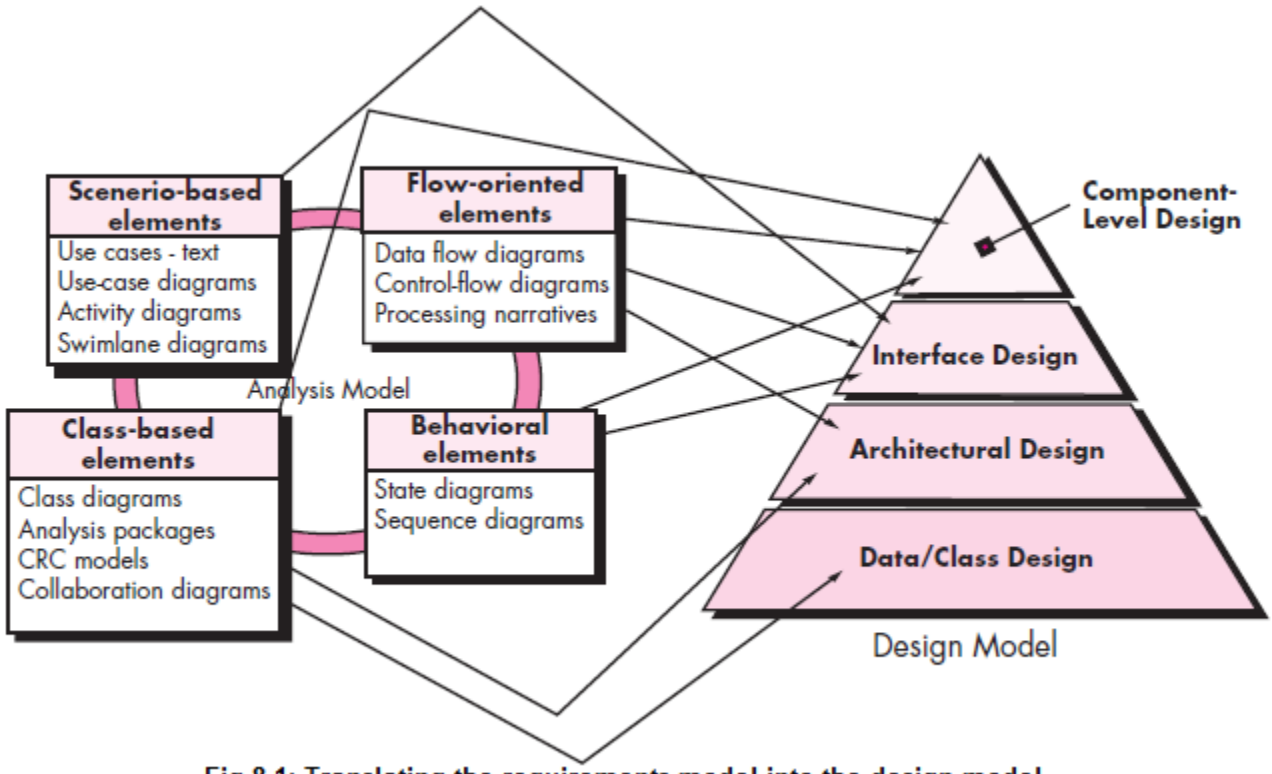

Fig 8.1: Translating the requirements model into the design model

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 Figure 8.1. 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 objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture.

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 *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.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained.

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. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

3.2 THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

3.2.1 Software Quality Guidelines and Attributes: Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. The 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, the software team must establish technical criteria for good design. Guide lines are as follows

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 (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes. 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. Early design work concentrated on criteria for the development of modular programs

and methods for refining software structures in a topdown manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods, model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

A number of design methods, growing out of the work just noted, are being applied throughout the industry. 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

Design concepts has evolved over the history of software engineering. Each concept provides the software designer with a foundation from which more sophisticated design methods can be applied. A brief overview of important software design concepts that span both traditional and object-oriented software development is given below.

3.3.1 Abstraction: When you consider a modular solution to any problem, 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. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

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*".
In its simplest form, *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.
One goal of software design is to derive an architectural rendering of a system. A set of architectural patterns enables a software engineer to solve common design problems.

Shaw and Garlan describe a set of properties 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. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
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.

3.3.3 Patterns: 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. Stated
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)
(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.

For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem. It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to *a divide-and-conquer strategy*—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

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 modules, that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". The

number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.
if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 8.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.

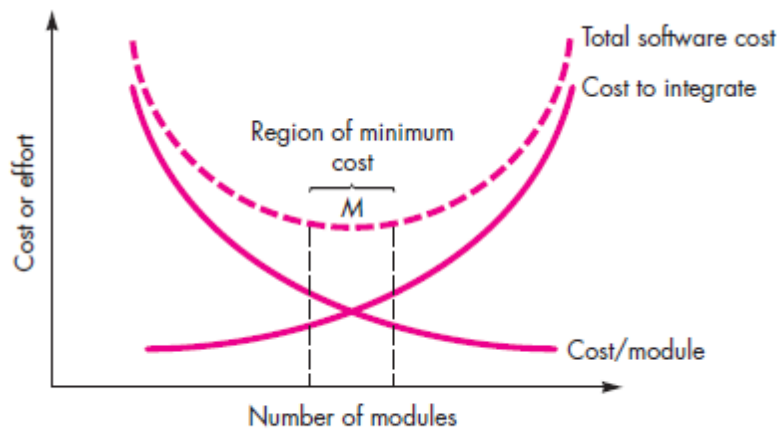

Fig 8.2: Modularity and software cost

Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

The curves shown in Figure 8.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. *Undermodularity* or *overmodularity* should be avoided.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

3.3.6 Information Hiding: The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that

communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. 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 "singleminded" function and an "aversion" to excessive interaction with other modules. Stated another way, *you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure*. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

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.

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 (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions.

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. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagate throughout a system.

3.3.8 Refinement: Stepwise refinement is a *top-down design strategy.* . A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a *macroscopic statement* of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of elaboration begins with a statement of function (or description of information) that is defined at a high level of abstraction. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs. Refinement helps you to reveal low-level details as design progresses.

3.3.9 Aspects: As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts". Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account".

3.3.10 Refactoring: An important design activity for many agile methods is *refactoring* a *reorganization technique* that simplifies the design (or code) of a component without changing its function or behavior. "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."
When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The result will be software that is easier to integrate, easier to test, and easier to maintain.

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: As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
Five different types of design classes, 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). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine), and 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. 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.

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 (based on a knowledgeable interpretation of the class name) to exist for the class.
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. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter*, suggests that a method should only send messages to methods in neighboring classes.

3.4 THE DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in Figure 8.4. 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. Referring to Figure 8.4, the dashed line indicates the boundary between *the analysis and design models*. The analysis model slowly blends into the design and a clear distinction is less obvious.

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

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. The *deployment model* is usually delayed until the design has been fully developed.

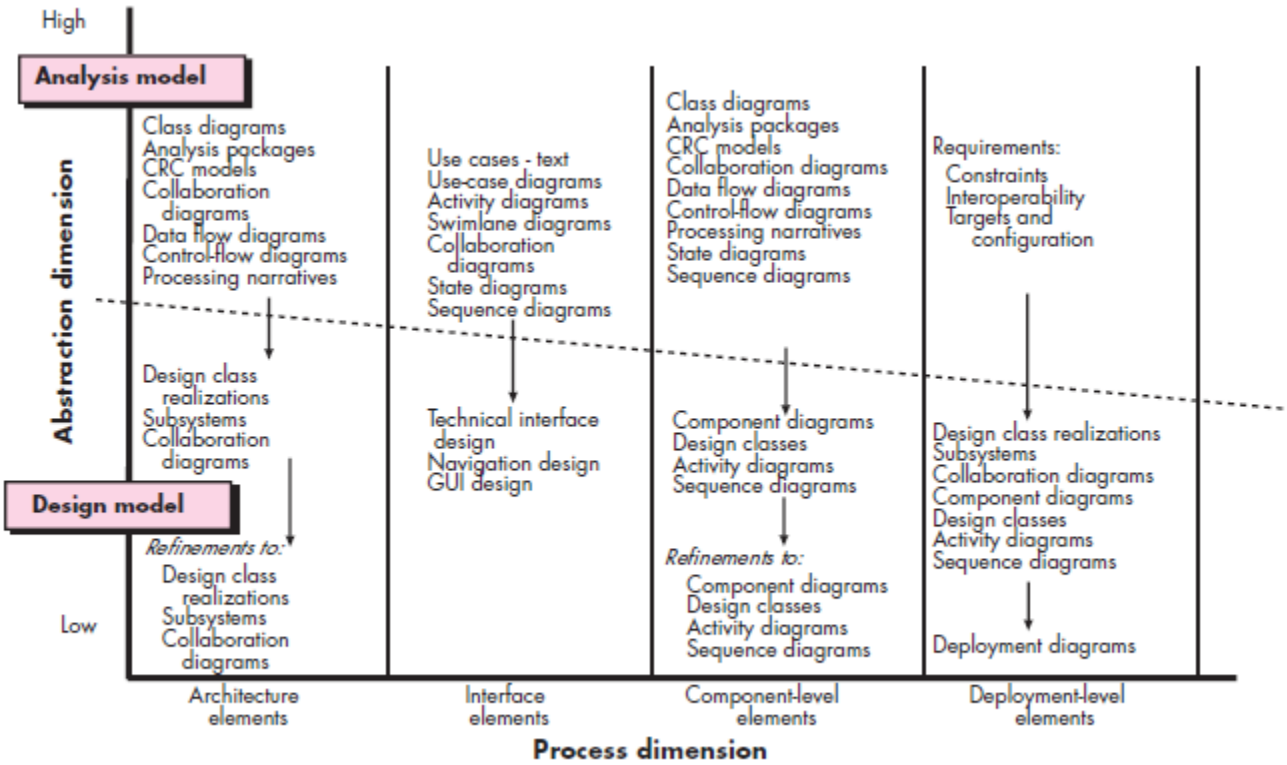

Fig 8.4: Dimensions of the design model

3.4.1 Data Design Elements: Like other software engineering activities, 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. 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.

3.4.2 Architectural Design Elements: The architectural design for software is the equivalent to the floor plan of a house. The floor plan gives us an overall view of the house. Architectural

design elements give us an overall view of the software. The architectural model is derived from three sources:
(1) information about the application domain for the software to be built;
(2) specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and
(3) the availability of architectural *styles* and *patterns*.
The architectural design element is usually depicted as a set of interconnected subsystems, Each subsystem may have it's own architecture

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. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
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.

UI design (increasingly called usability design) is a major software engineering action. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components).

The design of *external interfaces* requires definitive information about the entity to which information is sent or received. The design of *internal interfaces* is closely aligned with component-level design.

3.4.4 Component-Level Design Elements: The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors). Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 8.6.
A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or some other diagrammatic form (e.g.,

flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

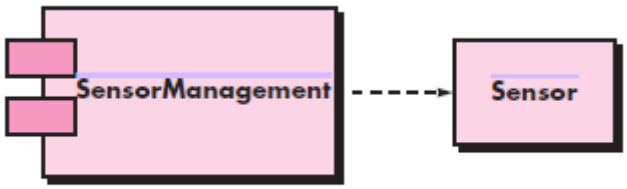

Fig 8.6: A UML component diagram

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.

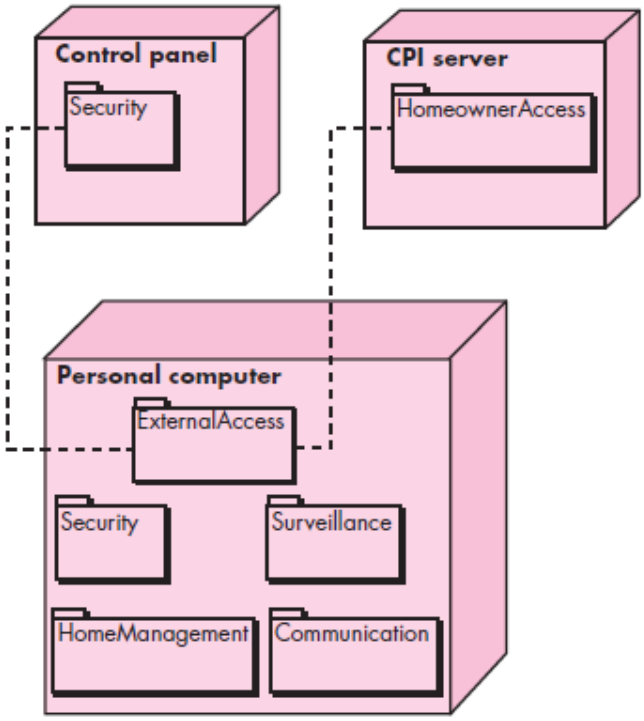

Fig 8.7: A UML deployment diagram

During design, a UML deployment diagram is developed and then refined as shown in Figure 8.7. The diagram shown is in descriptor form. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details.

3.5 SOFTWARE ARCHITECTURE

(Architecture → Architecture Style → Architecture Pattern → Design)

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.
Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements derived during software requirements analysis.
Why is it important? It provides you with the big picture and ensures that you've got it right.
What are the steps? Architectural design begins with *data design* and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.
What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.
How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

3.5.1 What Is Architecture? : Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another.

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) architectural alternatives at a stage when making design changes is still relatively easy
(3) reduce the risks associated with the construction of the software.
This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a

program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

There is a distinct difference between the terms *architecture* and *design*. A design is an instance of an architecture similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. *Architectural design focuses on the representation of the structure of software components, their properties, and interactions.*

3.5.1.2 Why Is Architecture Important?: 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".

3.5.1.3 Architectural Descriptions: Different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

Tyree and *Akerman* note this when they write: "Developers want clear, decisive guidance on how to proceed with design. Customers want a clear understanding on the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture's key aspects." Each of these "wants" is reflected in a different view represented using a different viewpoint.

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 [stakeholder] concerns."

3.5.1.4 Architectural Decisions: Each view developed as part of an architectural description addresses a specific stakeholder concern. *To develop each view 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.

3.5.2 ARCHITECTURAL GENRES

Although the underlying principles of architectural design apply to all types of architecture, *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. For example, within the genre of buildings, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.

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 (C4I) 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.

SAI (Software Architecture for Immersipresence) is a new software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. The goal of SAI is to provide a universal framework for the distributed implementation of algorithms and their easy integration into complex systems.

3.6 ARCHITECTURAL STYLES

Architectural 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.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways:
(1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;
(2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency)
(3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).
Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

3.6.1 A Brief Taxonomy of Architectural Styles: Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number 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. Figure 9.1 illustrates a typical data-centered style. Client

software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

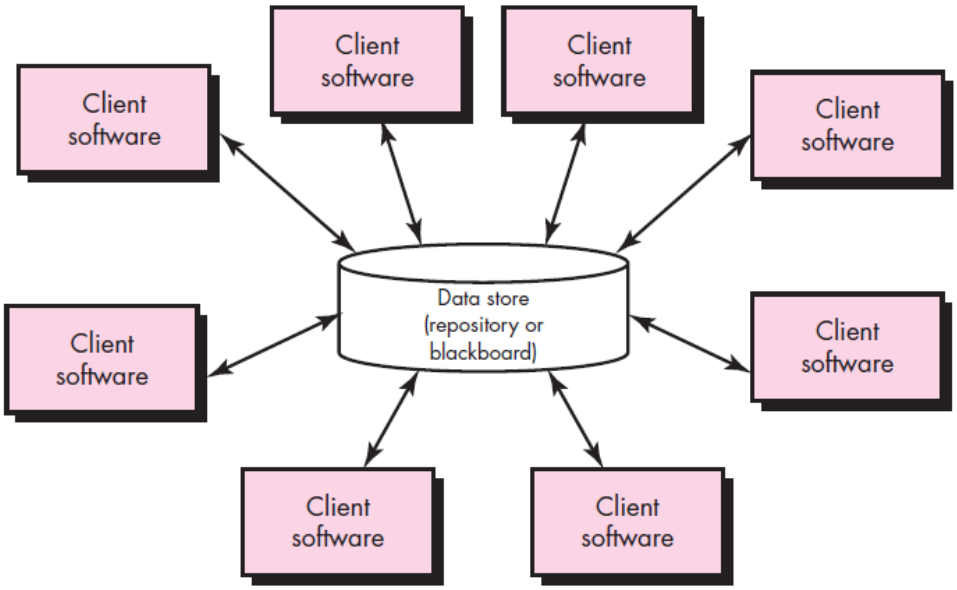

Fig 9.1: Data-centered architecture

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 (Figure 9.2) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

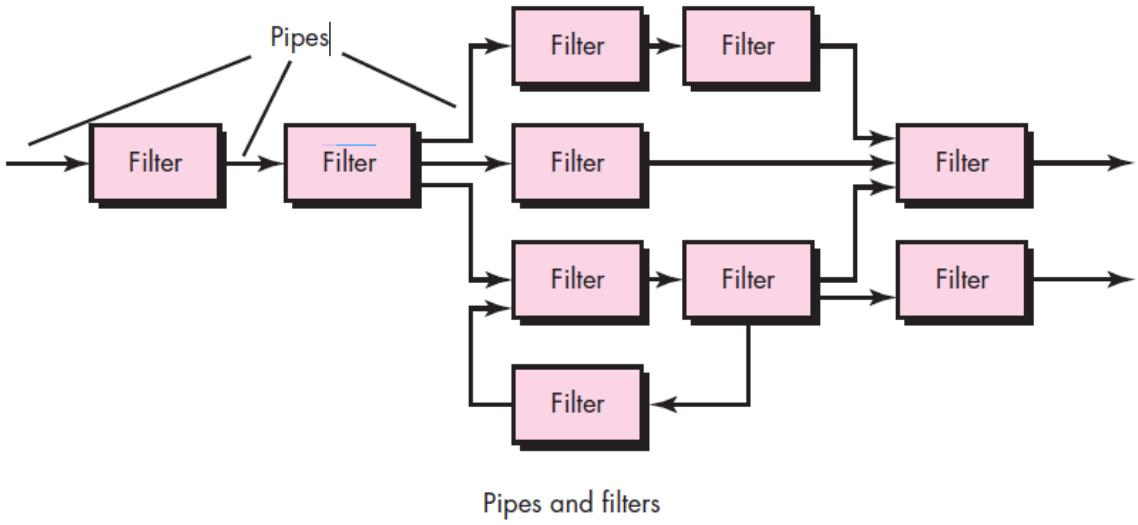

Fig 9.2: Data-flow architecture

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 substyles exist within this category:

• *Main program/subprogram architectures*. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Figure 9.3 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.

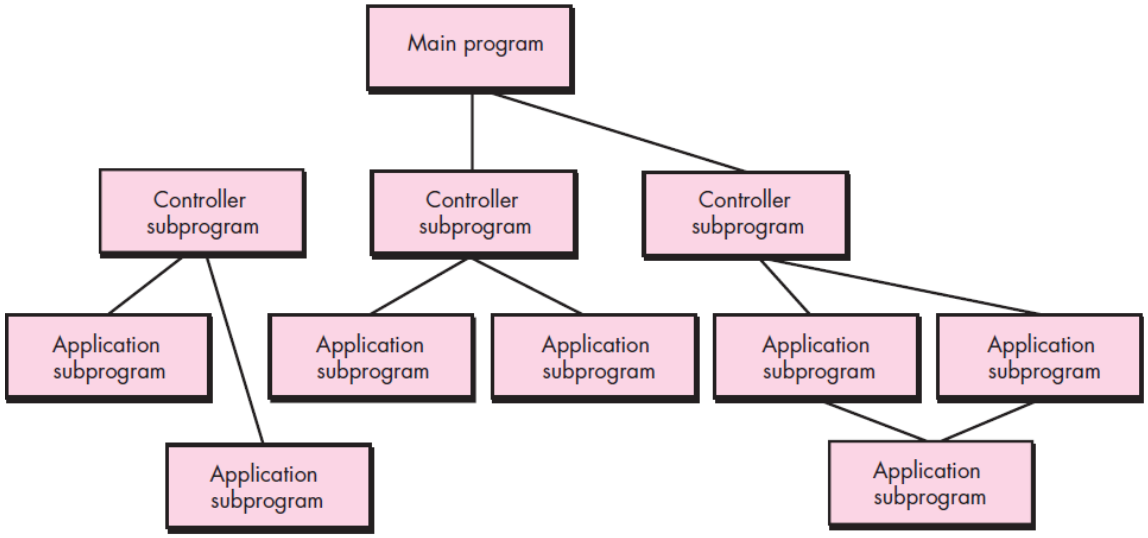

Fig 9.3: Main program / subprogram architecture

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.

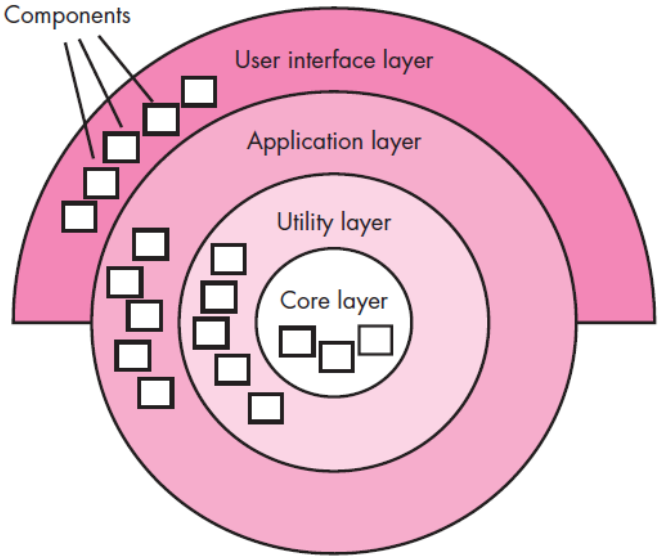

Fig 9.4: Layered architecture

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 9.4. 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. These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

3.6.2 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.

For example, the overall architectural style for an application might be call-and-return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns.

3.6.3 Organization and Refinement: Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. 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 (i.e., the geometric form that the control takes)? 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 (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? 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.

3.7 ARCHITECTURAL DESIGN

The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. *An archetype is an abstraction (similar to a class) that represents one element of system behavior*. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

3.7.1 Representing the System in Context: *Architectural context represents how the software interacts with entities external to its boundaries*. 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 Figure 9.5. Referring to the figure, systems that interoperate with the target system are represented as
• *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
• *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

• *Peer-level systems*—those systems that interact on a peer-to-peer basis (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. Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

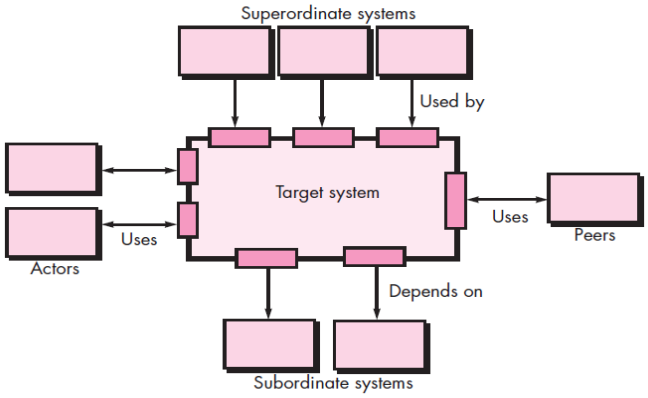

Fig 9.5: Architectural context diagram

3.7.2 Defining Archetypes: *Archetypes are the abstract building blocks of an architectural design*. 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 are the archetypes for safeHome: Node, Detector, Indicator., Controller.

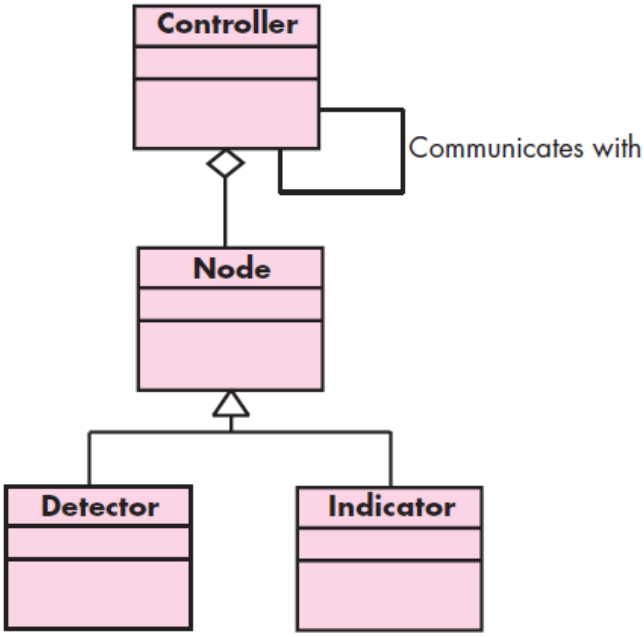

Fig 9.7: UML relationships for safehome function Archetypes

3.7.3 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. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

As an example for SafeHome home security, the set of top-level components that address the following functionality:

• *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

• *Control panel processing*—manages all control panel functionality.

• *Detector management*—coordinates access to all detectors attached to the system.

• *Alarm processing*—verifies and acts on all alarm conditions.

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

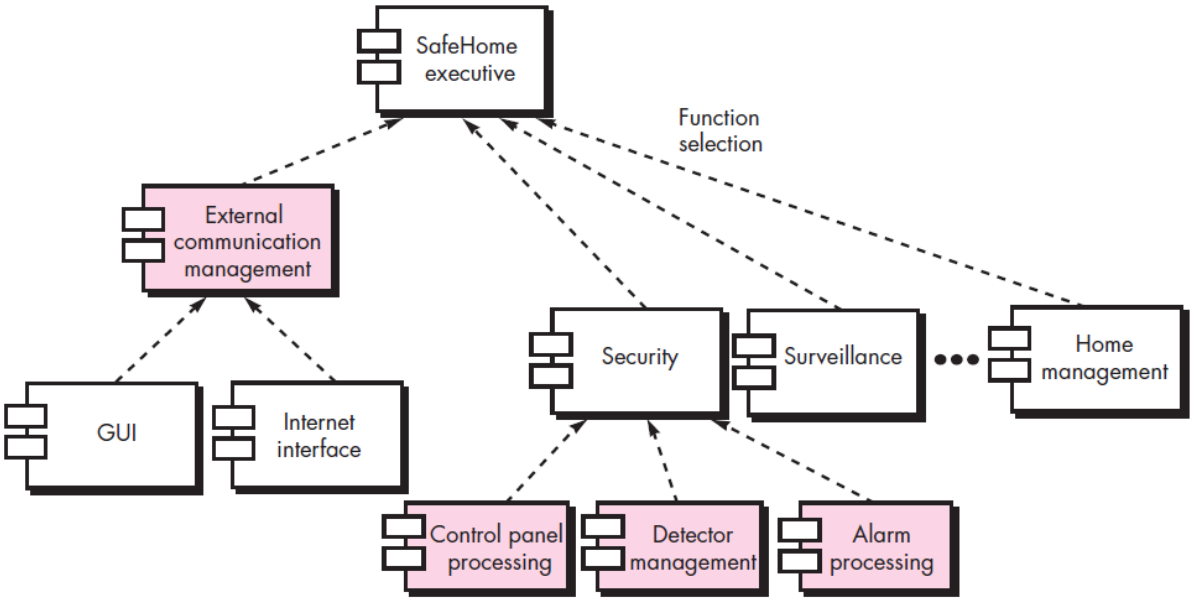

Fig 9.8: Overall architectural structure for *SafeHome* with top-level components

3.7.4 Describing Instantiations of the System: The architectural design that has been modeled to this point is still relatively *high level*. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, *further refinement* is still necessary.

3.8 ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. Two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

3.8.1 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* using the sensitivity analysis conducted in step 5.
The SEI describes this approach in the following manner.
Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.

3.8.2 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.
The three types of dependencies:
Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.
Flow dependencies represent dependence relationships between producers and consumers of resources. For example, for two components u and v, if u must complete before control flows into v (prerequisite), or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.
Constrained dependencies represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, u and v cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between u and v.

3.8.3 Architectural Description Languages: Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, language based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

3.9 ARCHITECTURAL MAPPING USING DATA FLOW

There is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in using the techniques.

To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems. The call and return architecture can reside within other more sophisticated architectures. For example, the architecture of one or more components of a client-server architecture might be call and return. 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.

3.9.1 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.

Step 1. Review 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. Figure 9.10 depicts a level 0 context model, and Figure 9.11 shows refined data flow for the security function.

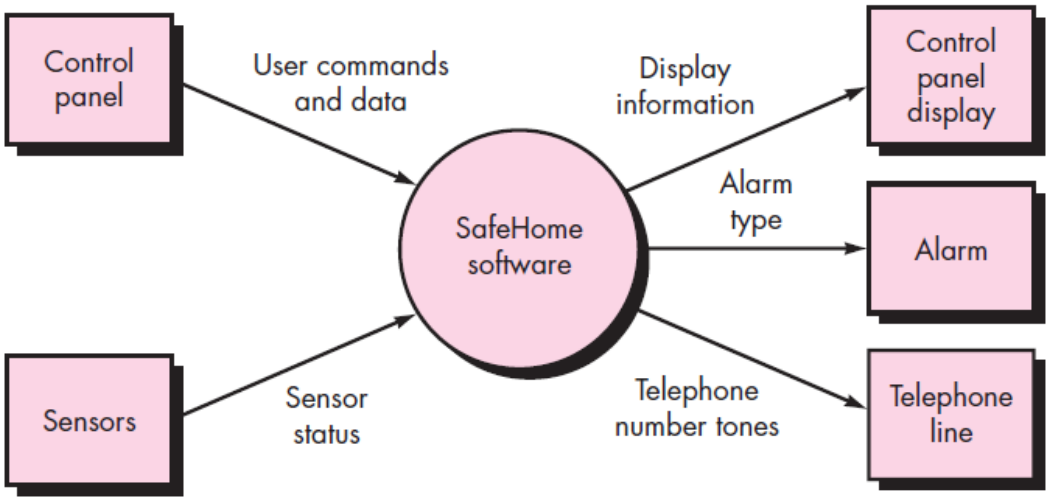

Fig 9.10 Context-level DFD for safeHome security function

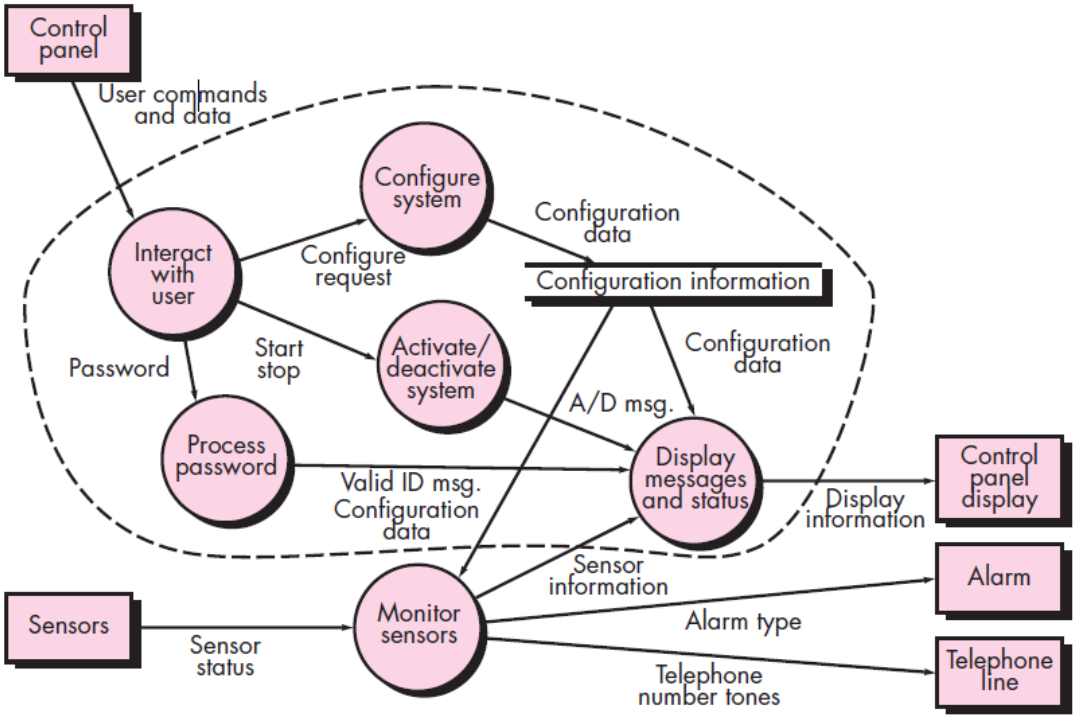

Fig 9.11: Level 1 DFD for safeHome security function

Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors (Figure 9.12) is examined, and a level 3 data flow diagram is derived as shown in Figure 9.13. The data flow diagram exhibits relatively high cohesion.

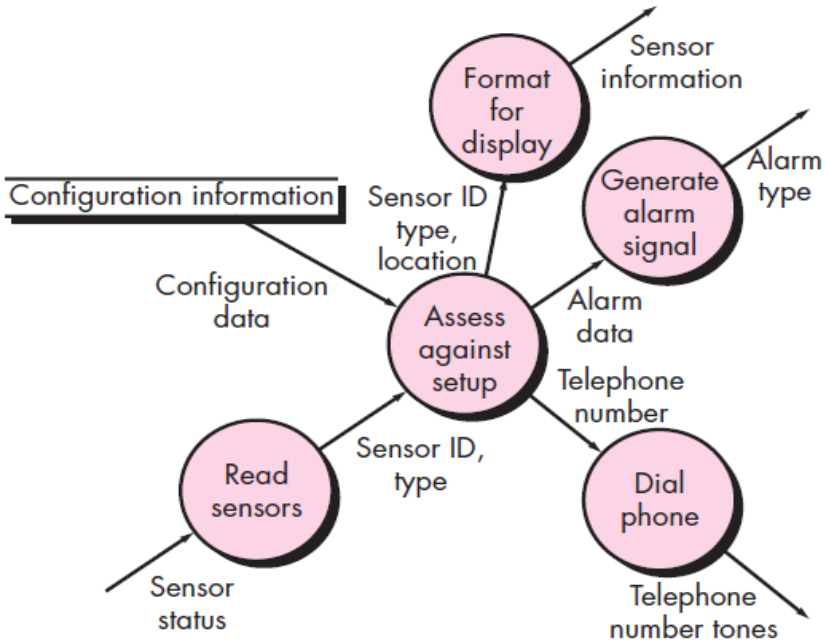

Fig 9.12: Level 2 DFD that refines the monitor sensors transform

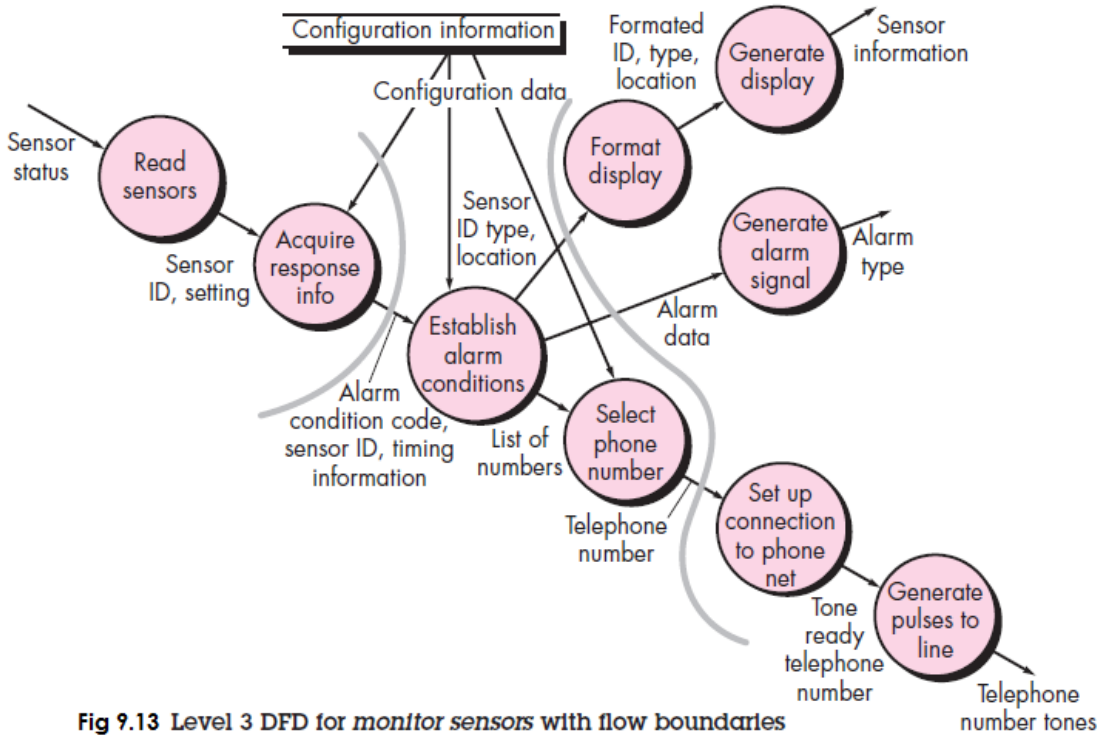

Fig 9.13 Level 3 DFD for *monitor sensors* with flow boundaries

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Evaluating the DFD (Figure 9.13), 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. Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 9.13. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary. The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

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.

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.

3.9.2 Refining the Architectural Design: Refinement of software architecture during early stages of design is to be encouraged. Alternative architectural styles may be derived, refined, and evaluated for the "best" approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.
It is important to note that structural simplicity often reflects both elegance and efficiency. 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.

3.10 Component-Level Design

What is it? Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.
Who does it? A software engineer performs component-level design.
Why is it important? You have to be able to determine whether the software will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with other design representations (i.e., the data, architectural, and interface designs). It provides a means for assessing whether data structures, interfaces, and algorithms will work.
What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The class definition or processing narrative for each component is translated into a detailed design that makes use of diagrammatic or text-based forms that specify internal data structures, local interface detail, and processing logic. Design notation encompasses UML diagrams and supplementary forms. Procedural design is specified using a set of structured programming constructs. It is often possible to acquire existing reusable software components rather than building new ones.
What is the work product? The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.
How do I ensure that I've done it right? A design review is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct and will produce the appropriate data or control transformation allocated to the component during earlier design steps.

3.10 WHAT IS A COMPONENT?: A component is a modular building block for computer software. More formally, component is *"a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."*

Components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

3.10.1 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.

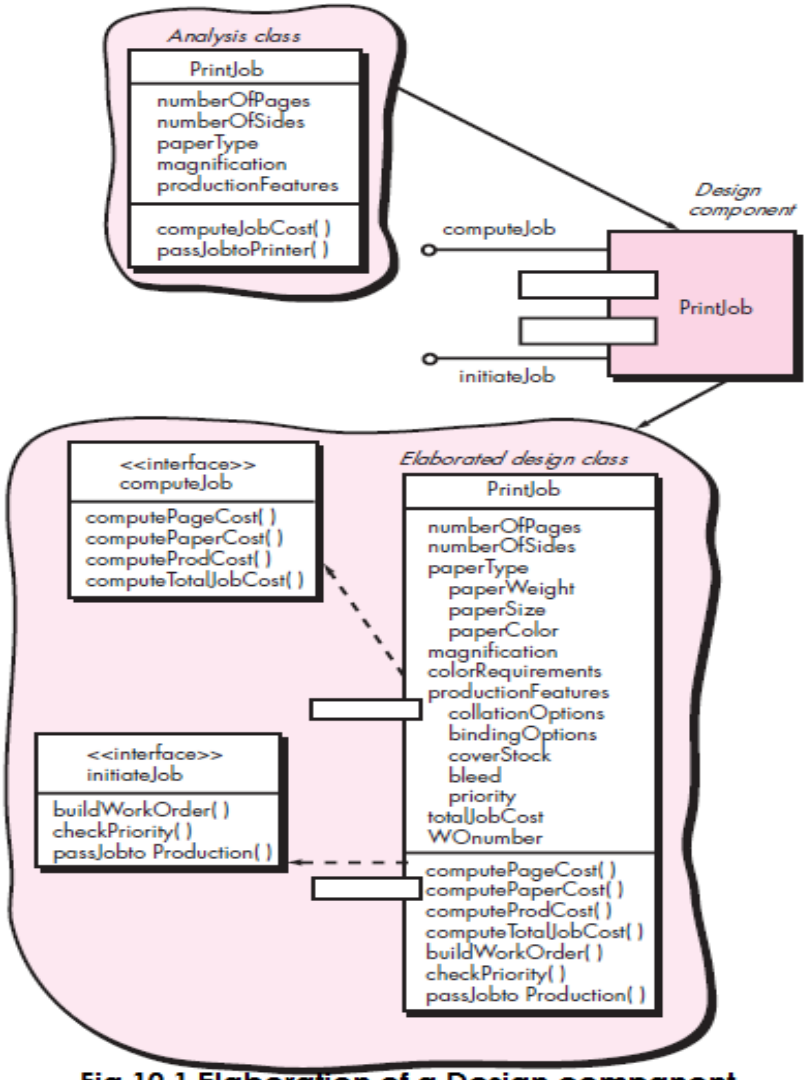

Fig 10.1 Elaboration of a Design companent

3.10.2 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.

Like object-oriented components, traditional software components are derived from the analysis model. To achieve effective modularity, design concepts like functional independence are applied as components are elaborated.

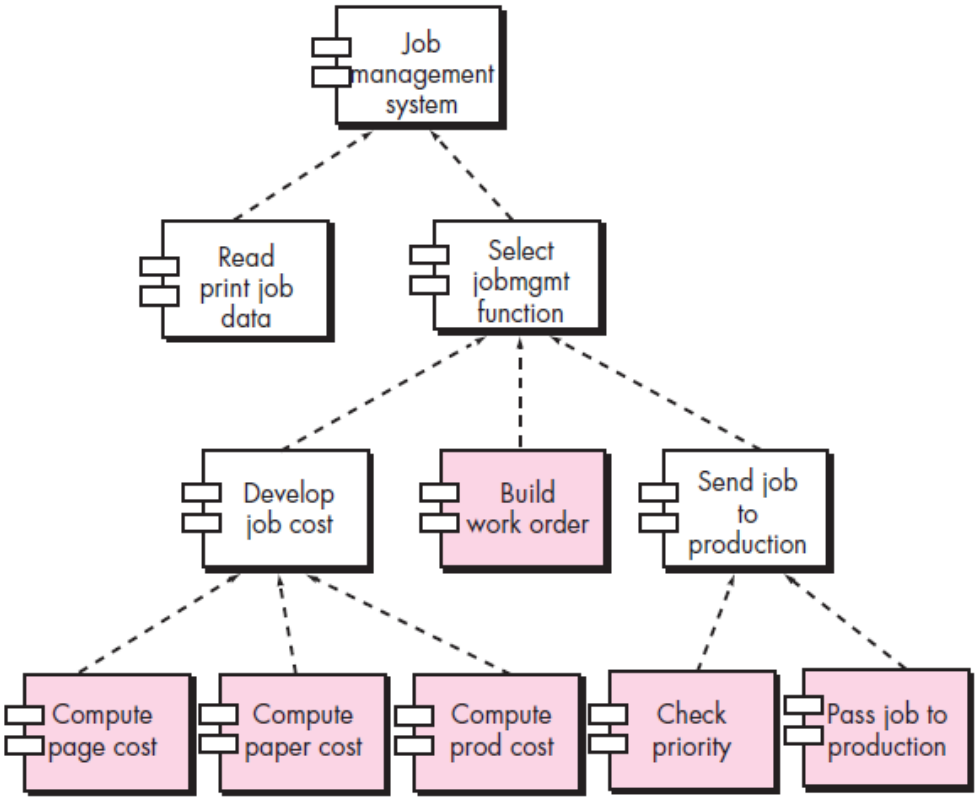

Fig 10.2: Structure chart for a traditional system

During component-level design, each module in Figure 10.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise

refinement approach. The behavior of the module is sometimes represented using a state diagram. Figure 10.3 represents the component-level design using a modified UML notation.

3.10.3 A Process-Related View: The object-oriented and traditional views of component-level design assume that the component is being designed from scratch. That is, you have to create a new component based on specifications derived from the requirements model.

Over the past two decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. In essence, a catalog of proven design or code-level components is made available to you as design work proceeds. As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture.

Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available.

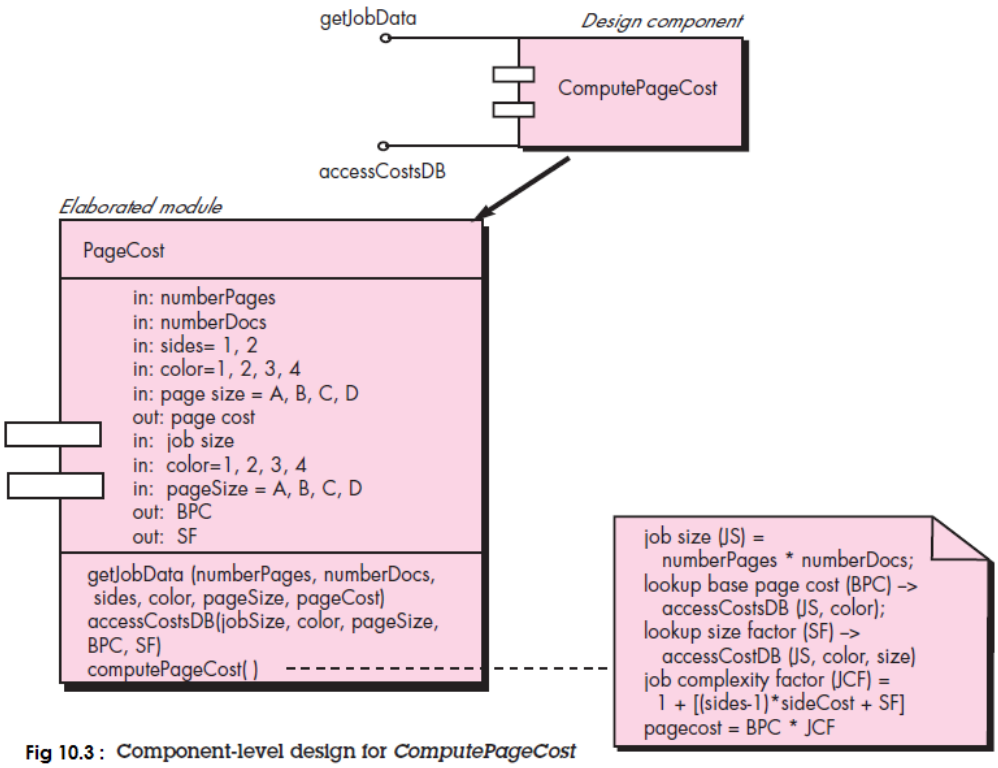

Fig 10.3 : Component-level design for *ComputePageCost*

3.11 DESIGNING CLASS-BASED COMPONENTS

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

3.11.1 Basic Design Principles: Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied.
The 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 (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

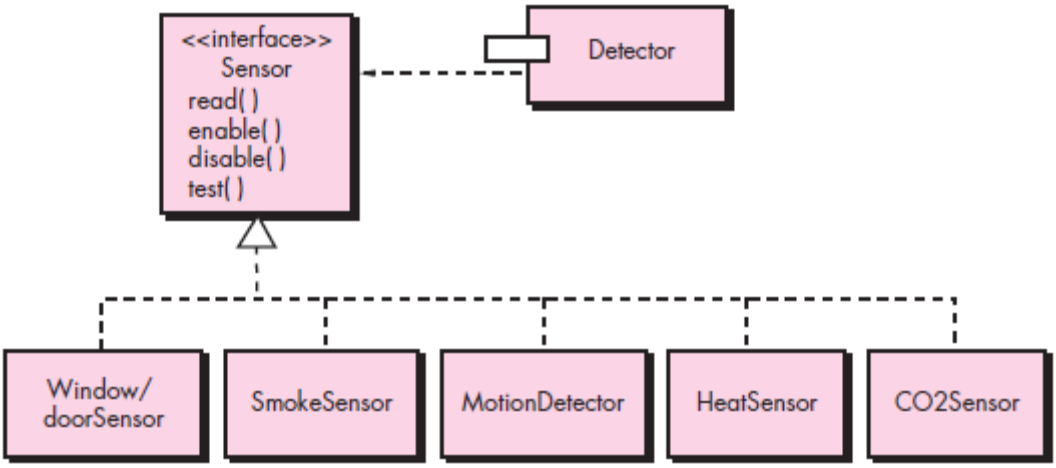

Fig 10.4: Following the OCP

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.

Dependency Inversion Principle (DIP). "Depend on abstractions. Do not depend on concretions". As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other

concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP). "Many client-specific interfaces are better than one general purpose interface". There are many instances in which multiple client components use the operations provided by a server class. *ISP suggests that you should create a specialized interface to serve each major category of clients.* 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.

Martin suggests additional *packaging* principles that are applicable to component-level design:
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. 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" If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

3.11.2 Component-Level Design Guidelines: Ambler suggests the following guidelines for components, their interfaces, and the dependencies and inheritance characteristics
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. However, unfettered representation of interfaces tends to complicate component diagrams. Ambler recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex; (2) for consistency, interfaces should flow from the left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.

Dependencies and Inheritance. For improved readability, it is a good to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). In addition, component interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency.

3.11.3 Cohesion: Cohesion is the "single-mindedness" of a component. Within the context of component-level design for object-oriented systems, cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. Lethbridge and Laganiére define a number of different types of cohesion.

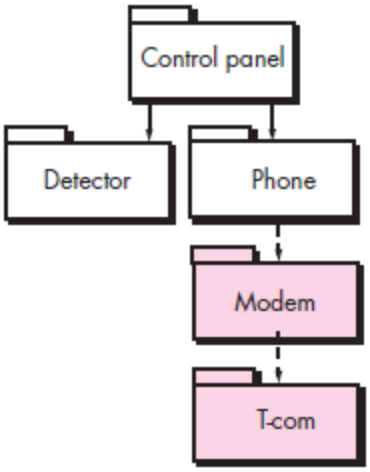

Fig 10.5: Layer cohesion

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. It might be possible to define a set of layered packages as shown in Figure 10.5. The shaded packages contain infrastructure components.
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. Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.

3.11.4 Coupling: Communication and collaboration are essential elements of any object-oriented system. *Coupling is a qualitative measure of the degree to which classes are connected toone 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.
Lethbridge and Laganiére define the following coupling categories:
Content coupling. Occurs when one component "surreptitiously modifies data that is internal to another component". This violates information hiding—a basic design concept.

Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), 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 (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class for its type". If the type definition changes, every component that uses the definition must also change.

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

External coupling. Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). 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 and understand the ramifications of high coupling when it cannot be avoided.

3.11.5 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. As we have noted earlier, classes and components in this category include GUI components, operating system components, and object and data management components.

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.

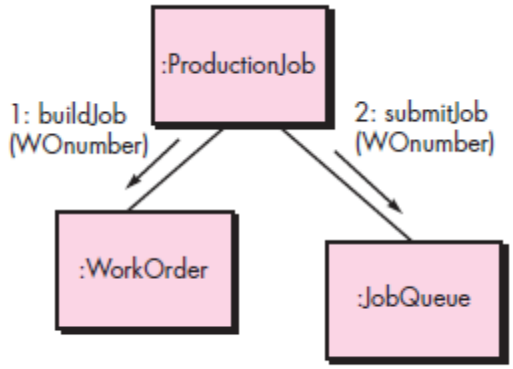

Fig 10.6: Collaboration diagram with messaging

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*. Figure 10.6 illustrates a simple collaboration diagram for the printing system.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. *Interface is the equivalent of an abstract class that provides a controlled connection between design classes*.

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 used for implementation. UML defines an attribute's data type using the following syntax:

name : *type-expression* _ *initial-value {property string}*

where *name* is the attribute name, *type expression* is the data type, initial value is the value that the attribute takes when an object is created, and *property-string* defines a property or characteristic of the attribute.

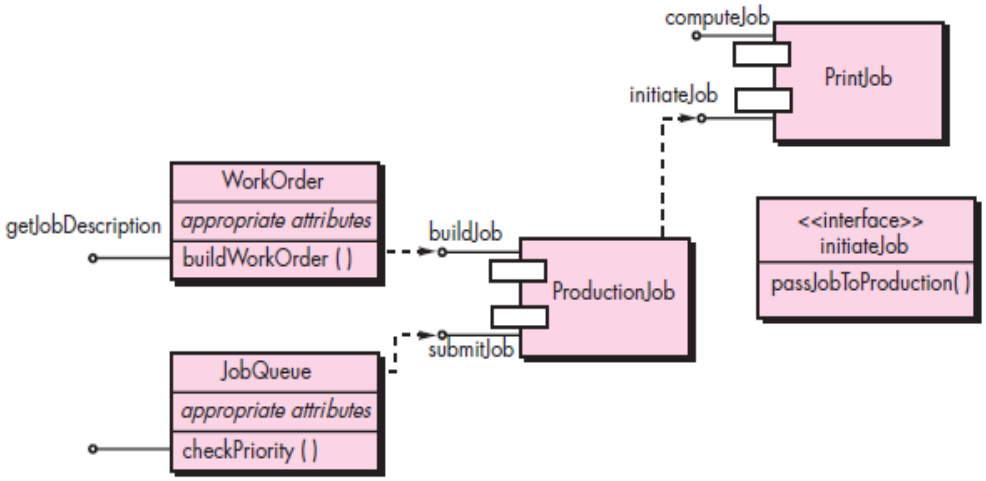

Fig 10.7: Refactoring interfaces and class definitions for PrintJob

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.

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name. Figure 10.8 depicts a UML activity diagram for *computePaperCost().*

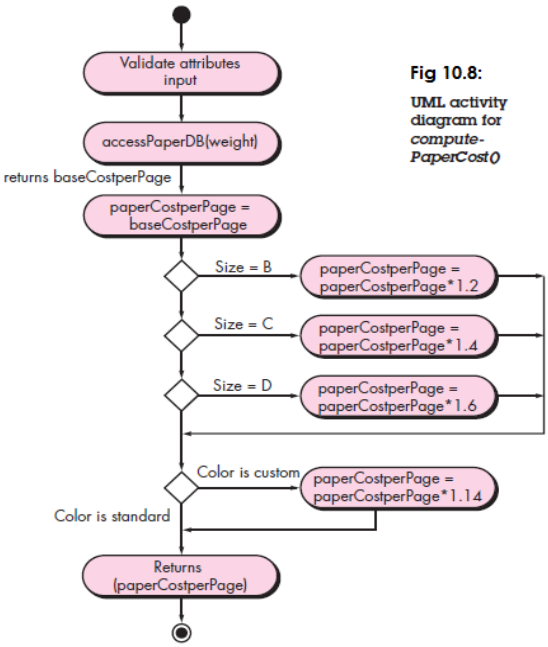

Fig 10.8: UML activity diagram for *compute-PaperCost()*

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. The dynamic behavior of an object is affected by events that are external to it and the current state of the object as illustrated in Figure 10.9.

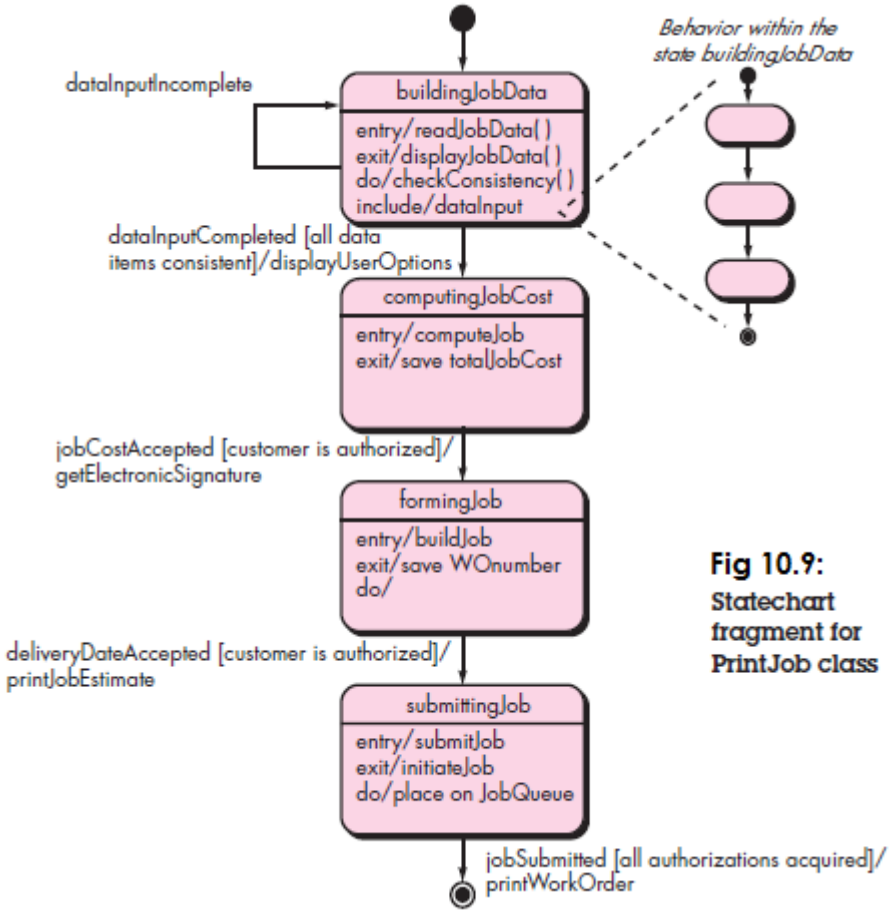

Fig 10.9: Statechart fragment for PrintJob class

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 are represented within the context of the computing environment that will house them.

Step 7. Refactor every component-level design representation and always consider alternatives. The design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the nth iteration you apply to the model. It is essential to refactor as design work is conducted. Develop alternatives and consider each carefully, using the design principles and concepts.

3.12 COMPONENT-LEVEL DESIGN FOR WEBAPPS

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.

3.12.1 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.

The formality of *content design* at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity grows, it may be necessary to organize content in a way that allows easier reference and design manipulation. In addition, if *content is highly dynamic*, it becomes important to establish a clear structural model that incorporates content components.

3.12.2 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 these (and many other) capabilities, you will design and construct WebApp functional components that are similar in form to software components for conventional software. 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.

3.13 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 emphasized "*maintenance of functional domain.*" That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

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. The structured constructs were proposed to limit the procedural design of software to a small number of predictable logical structures.

Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances *readability*, *testability*, and *maintainability*.

The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

3.13.1 Graphical Design Notation: ”A picture is worth a thousand 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—all elements of structured programming—and is a descendent of an earlier pictorial design representation called a flowchart. A *box* is used to indicate a *processing* step. A *diamond* represents a *logical condition*, and *arrows* show the *flow of control*. Figure 10.10 illustrates three structured constructs.

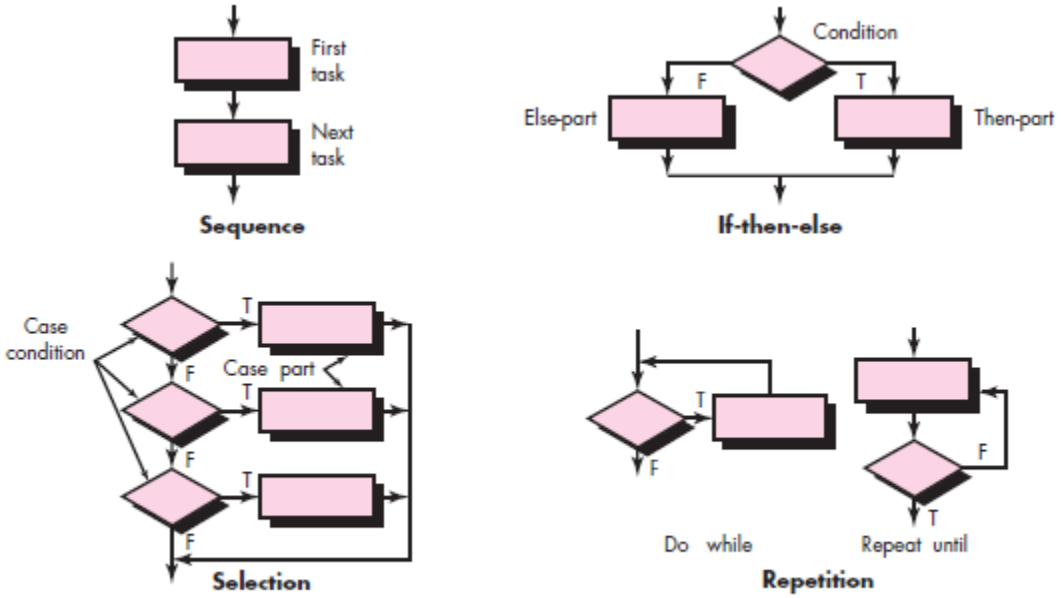

Fig 10.10: Flowchart constructs

The *sequence* is represented as two processing boxes connected by a line (arrow) of control. Condition, also called *if-then-else*, is depicted as a decision diamond that, if true, causes then-part processing to occur, and if false, invokes else-part processing. *Repetition* is represented using two slightly different forms. The *do while* tests a condition and executes a loop task

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

3.13.2 Tabular Design Notation: Decision tables provide a notation that translates actions and conditions (described in a processing narrative or a use case) 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 Figure 10.11. 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 10.11: Decision table nomenclature

3.13.3 Program Design Language: Program design language (PDL), also called structured English or pseudocode, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English). Narrative text (e.g., English) is embedded within a programming language-like syntax. Automated tools can be used to enhance the application of PDL.

A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs. It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, interprocess synchronization, and many other features. The application design for which PDL is to be used should dictate the final form for the design language. The format and semantics for some of these PDL constructs are presented in the example that follows.

```
do for all sensors
    invoke checkSensor procedure returning signalValue
    if signalValue > bound [alarmType]
        then phoneMessage = message [alarmType]
            set alarmBell to "on" for alarmTimeSeconds
            set system status = "alarmCondition"
            parbegin
                invoke alarm procedure with "on", alarmTimeSeconds;
                invoke phone procedure set to alarmType, phoneNumber
            endpar
        else skip
    endif
enddofor
end alarmManagement
```

3.14 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."

3.14.1 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.

It is important to note that domain analysis is applicable to any software engineering paradigm and may be applied for conventional as well as object-oriented development.

3.14.2 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. Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development actions is applied when a component is proposed for use.

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.

An interface description provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can, in fact, be reused effectively in a new 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 nonnumerical algorithms.
• Exception handling

Component Adaptation. In an ideal setting, domain engineering creates a library of components that can be easily integrated into an application architecture. The implication of "easy integration" is that (1) consistent methods of resource management have been implemented for all components in the library, (2) common activities such as data management exist for all components, and (3) interfaces within the architecture and with the external environment have been implemented in a consistent manner.

Conflicts may occur in one or more of the areas in selection of components. To avoid these conflicts, an adaptation technique called component wrapping is sometimes used. When a software team has full access to the internal design and code for a component *white-box* wrapping is applied. Like its counterpart in software testing 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 postprocessing 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 and coordination must be established to bind the components into an operational system.

OMG/CORBA. The Object Management Group has published a common object request broker architecture (OMG/CORBA). An object request broker (ORB) provides a variety of services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.

Microsoft COM and .NET. Microsoft has developed a component object model (COM) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. From the point of view of the application, "the focus is not on how implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects". The Microsoft .NET framework encompasses COM and provides a reusable class library that covers a wide array of application domains.

Sun JavaBeans Components. The JavaBeans component system is a portable, platform-independent CBSE infrastructure developed using the Java programming language. The JavaBeans component system encompasses a set of tools, called the Bean Development Kit (BDK), that allows developers to (1) analyze how existing Beans (components) work, (2) customize their behavior and appearance, (3) establish mechanisms for coordination and communication, (4) develop custom Beans for use in a specific application, and (5) test and evaluate Bean behavior.

3.14.3 Analysis and Design for Reuse: Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, software quality assurance (SQA), and correctness verification methods all contribute to the creation of software components that are reusable.

The requirements model is analyzed to determine those elements that point to existing reusable components. Elements of the requirements model are compared to WebRef descriptions of reusable components in a process that is sometimes referred to as "specification matching". If specification matching points to an existing component that fits the needs of the current

application, you can extract the component from a reuse library (repository) and use it in the design of a new system. If components cannot be found (i.e., there is no match), a new component is created i.e design for reuse (DFR) should be considered.

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 intramodular 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. Once standard data, interfaces, and program templates have been established, you have a framework in which to create the design. New components that conform to this framework have a higher probability for subsequent reuse.

3.14.4 Classifying and Retrieving Components: Consider a large component repository. Tens of thousands of reusable software components reside in it.

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 *content* of a component describes how the concept is realized. The *context* places a reusable software component within its domain of applicability.

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 (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.

• CBSE tools that support the integration of reused components into a new design or implementation.

Each of these functions interact with or is embodied within the confines of a reuse library.

The *reuse library* is one element of a larger software repository and provides facilities for the storage of software components and a wide variety of reusable work products (e.g., specifications, designs, patterns, frameworks, code fragments, test cases, user guides).

If an initial query results in a voluminous list of candidate components, the query is refined to narrow the list. Concept and content information are then extracted (after candidate components are found) to assist you in selecting the proper component.