➢ Design Engineering

Encompasses the set of principles, concepts and practices that lead to the development of high quality system or product. Design creates a representation or model of the software. Design model provides details about S/W architecture, interfaces and components that are necessary to implement the system. Quality is established during Design. Design should exhibit firmness, commodity and design. Design sits at the kernel of S/W Engineering. Design sets the stage for construction.

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, andcomponents that are necessary to implement the system.

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.

What is the work product?

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).

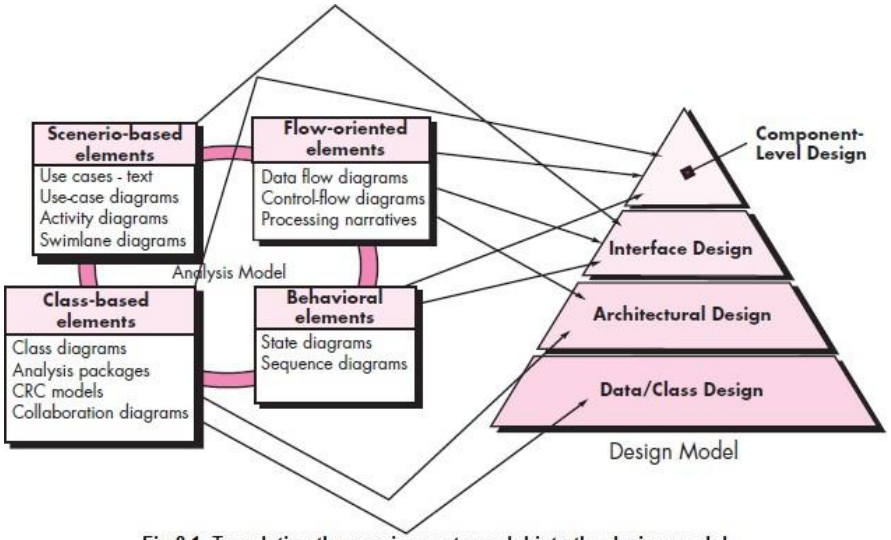


Fig 8.1: Translating the requirements model into the design model

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 architecturecan 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.

➢ THE DESIGN PROCESS AND QUALITY

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.

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.

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

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.

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

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

A design should lead to data structures that are appropriate for the classes to be implementedand are drawn from recognizable data patterns.

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

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

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

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

FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

Usability is assessed by considering human factors, overall aesthetics, consistency, and documentation.

Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

Performance is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

Supportability combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

The Evolution of Software Design: The evolution of software design is a continuing process. Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top down manner. Procedural aspects of design definition evolved into a philosophy called structured programming. 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.

➢ DESIGN CONCEPTS

Design concepts have 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.

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.

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.

Patterns: A pattern is a named suggest 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 isapplied and used.

The intent of each design pattern is to provide a description that enables a designer to determine

whether the pattern is applicable to the current work

whether the pattern can be reused (hence, saving design time)

whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns: Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.

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.

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.

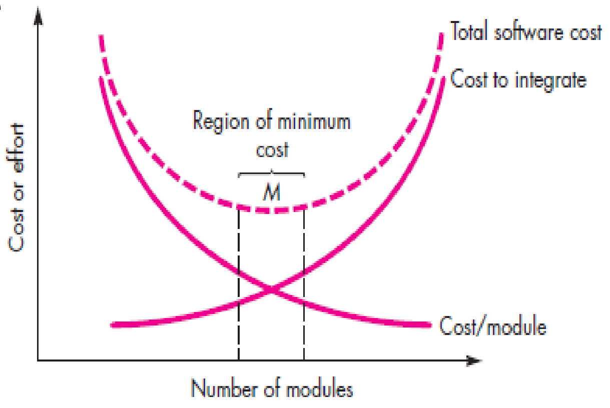


Fig 8.2: Modularity and software cost

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.

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.

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.

Functional Independence: The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules. 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 indicationof 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 propagates throughout a system.

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.

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" .

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.

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

Design Classes: 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.

➢ 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.

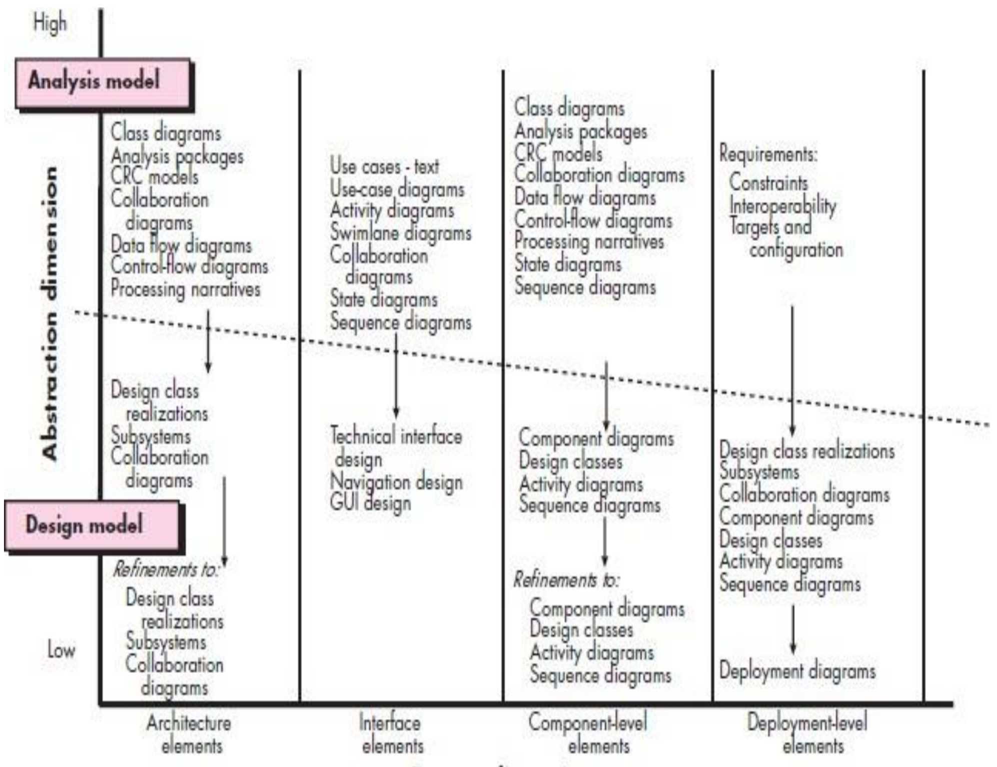


Fig 8.4: Dimensions of the design model

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.

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 fromthree sources:

Information about the application domain for the software to be built;

specific requirements model elements such as data flow diagrams or analysis classes, theirrelationships and collaborations for the problem at hand; and

the availability of architectural styles and patterns.

The architectural design element is usually depicted as a set of interconnected subsystems,

Each subsystem may have its own architecture

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:

the user interface (UI);

external interfaces to other systems, devices, networks, or other producers or consumers ofinformation; and

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.

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.

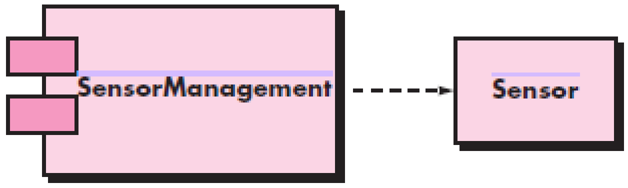


Fig 8.6: A UML component diagram

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.

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.

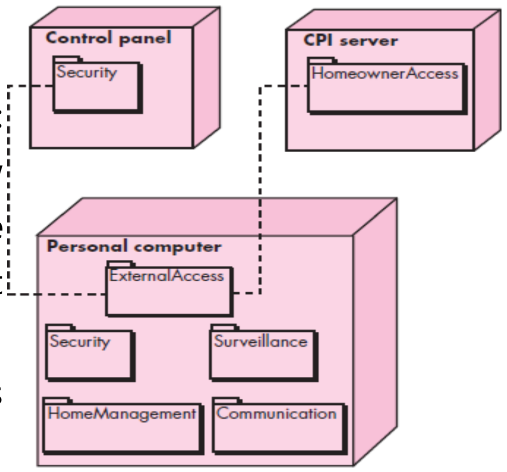


Fig 8.7: A UML deployment diagram

During design, a UML deployment diagram is developed and then refined as shown in Figures 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.

SOFTWARE ARCHITECTURE

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 analyze the effectiveness of the design in meeting its stated requirements architectural alternatives at a stage when making design changes is still relatively easy 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.

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" .

Architectural Descriptions: Different stakeholders will see 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.

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 .

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, orother organic processes.

Commercial and nonprofit—Systems that are fundamental to the operation of a businessenterprise.

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 multimediaartifacts.

Devices—Systems that interact with the physical world to provide some point service for anindividual.

Entertainment and sports—Systems that manage public events or that provide a large groupentertainment experience.

Financial—Systems that provide the infrastructure for transferring and managing money andother securities.

I. Games—Systems that provide an entertainment experience for individuals or groups.
II. Government—Systems that support the conduct and operations of a local, state, federal,global, or other political entity.
III. Industrial—Systems that simulate or control physical processes.
IV. Legal—Systems that support the legal industry.
V. Medical—Systems that diagnose or heal or that contribute to medical research.
VI. 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.

ARCHITECTURAL STYLES

Architectural style describes a system category that encompasses

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

a set of connectors that enable "communication, coordination and cooperation" among

components;

constraints that define how components can be integrated to form the system; and

semantic models that enable a designer to understand the overall properties of a system byanalyzing 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 architecture. However, a pattern differs from a style in a number of fundamental ways:

the scope of a pattern is less broad, focusing on one aspect of the architecture rather thanthe architecture in its entirety;

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)

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 asystem.

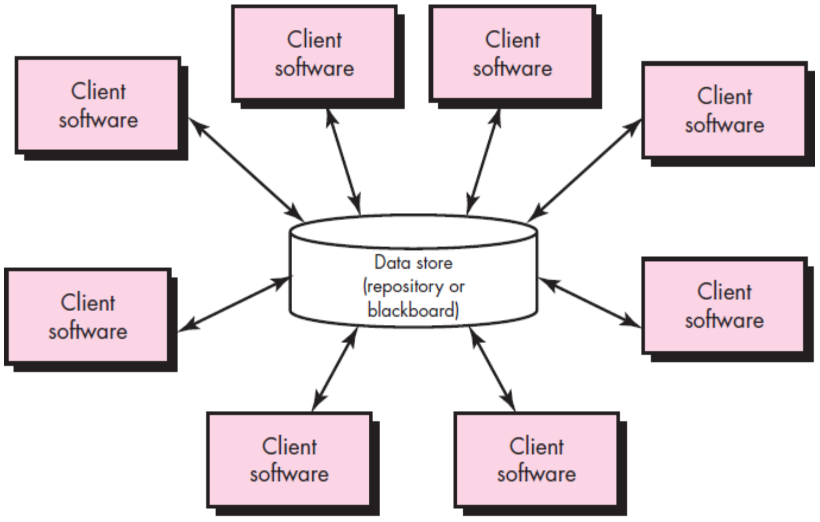


Fig 9.1: Data-centered architecture

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.

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.

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:

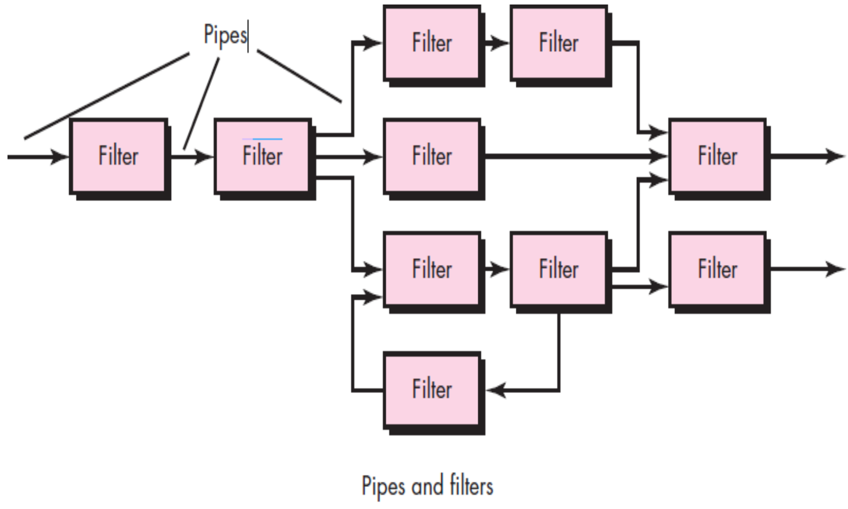


Fig 9.2: Data-flow architecture

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 architecture of this type.

Remote procedure call architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.

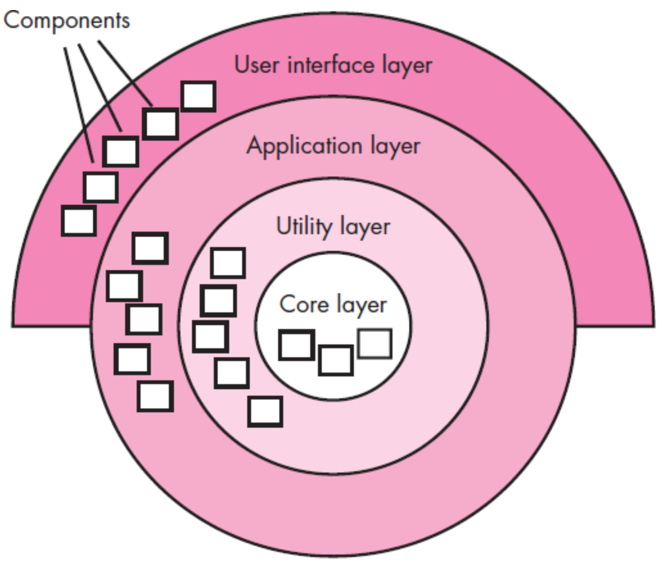


Fig 9.4: Layered architecture

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

Layered architectures. The basic structure of a layered architecture is

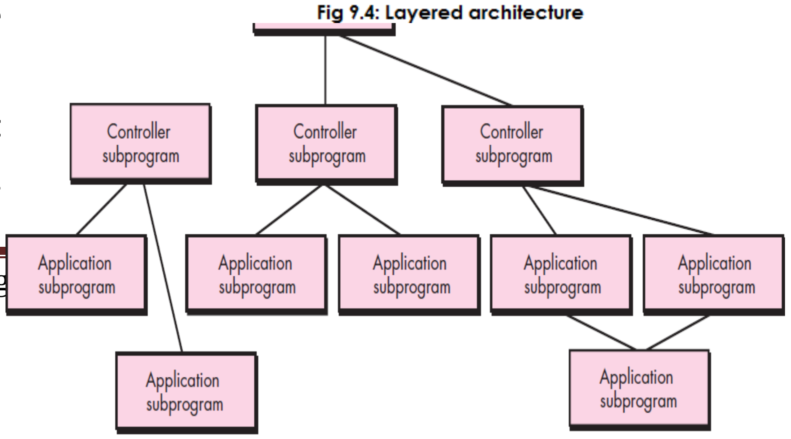


Fig 9.3: Main program / subprogram architecture

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.

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.

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.

➢ 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.

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

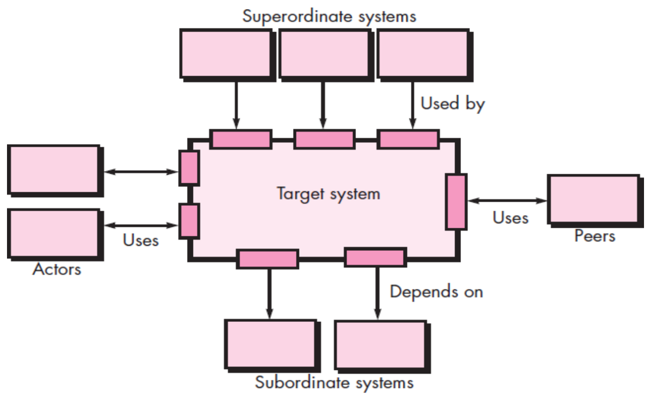


Fig 9.5: Architectural context diagram

Super ordinate systems—those systems that use the target system as part of some higher-levelprocessing 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).
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 instantiatedmany different ways based on the behavior of the system.

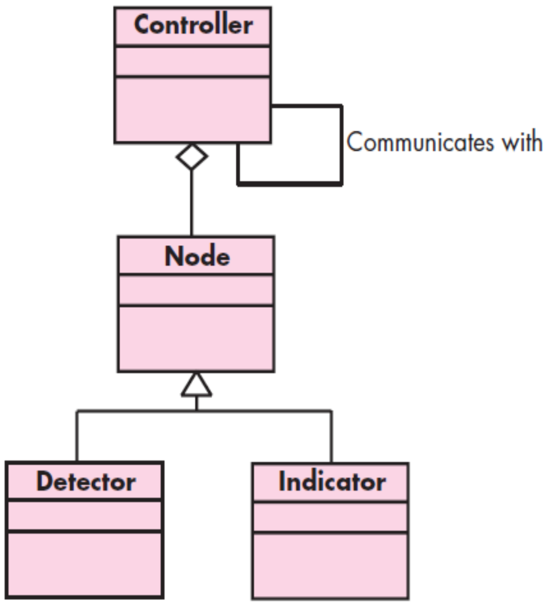


Fig 9.7: UML relationships for safehome function Archetypes

The following are the archetypes for safe Home: Node, Detector, Indicator., and Controller.

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.

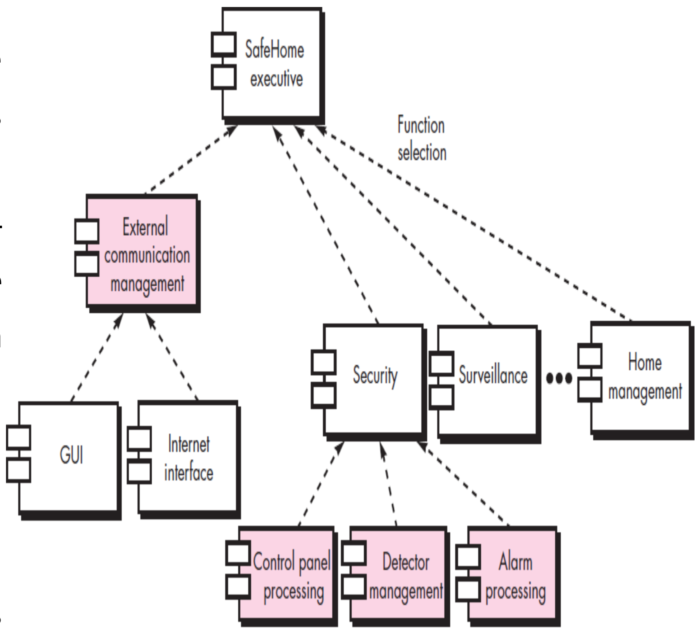


Fig 9.8: Overall architectural structure for *SafeHome* with top-level components

Alarm processing—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positionedwithin the overall Safe Home architecture.

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.

➢ ARCHITECTURAL MAPPING USING DATA FLOW

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.

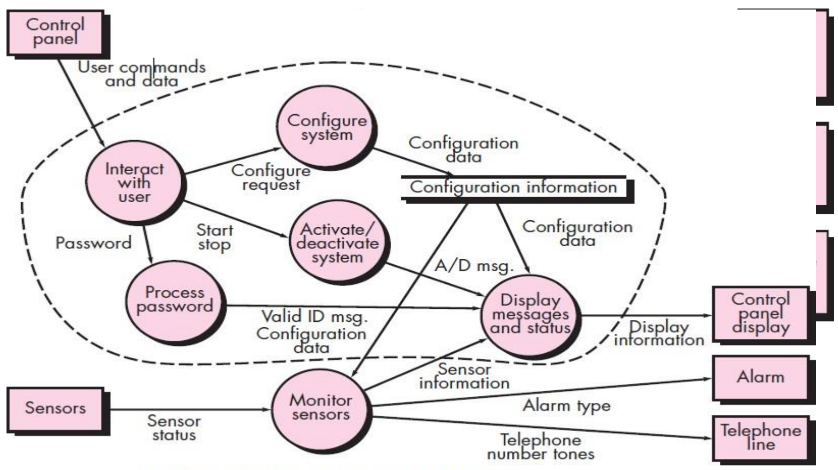


Fig 9.11: Level 1 DFD for safeHome security function

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.

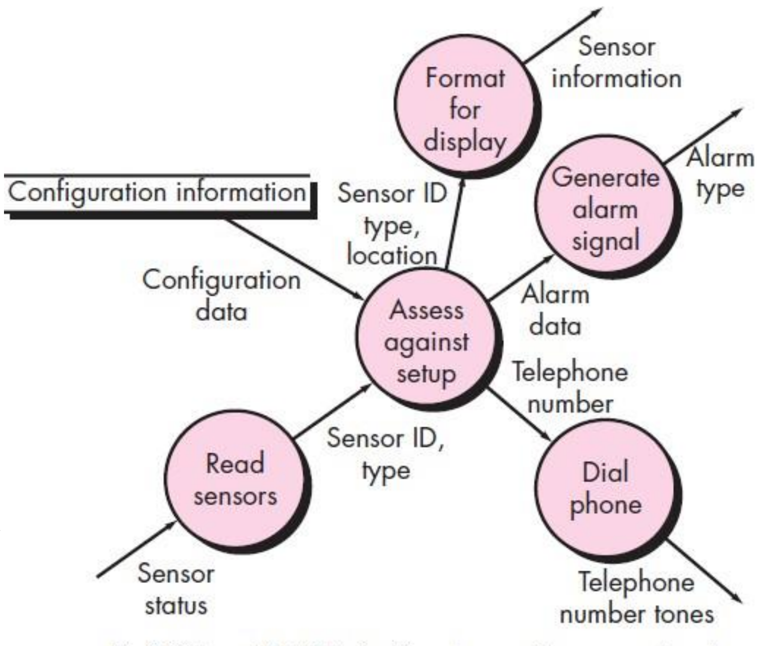


Fig 9.12: Level 2 DFD that refines the monitor sensors transform

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.

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.

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

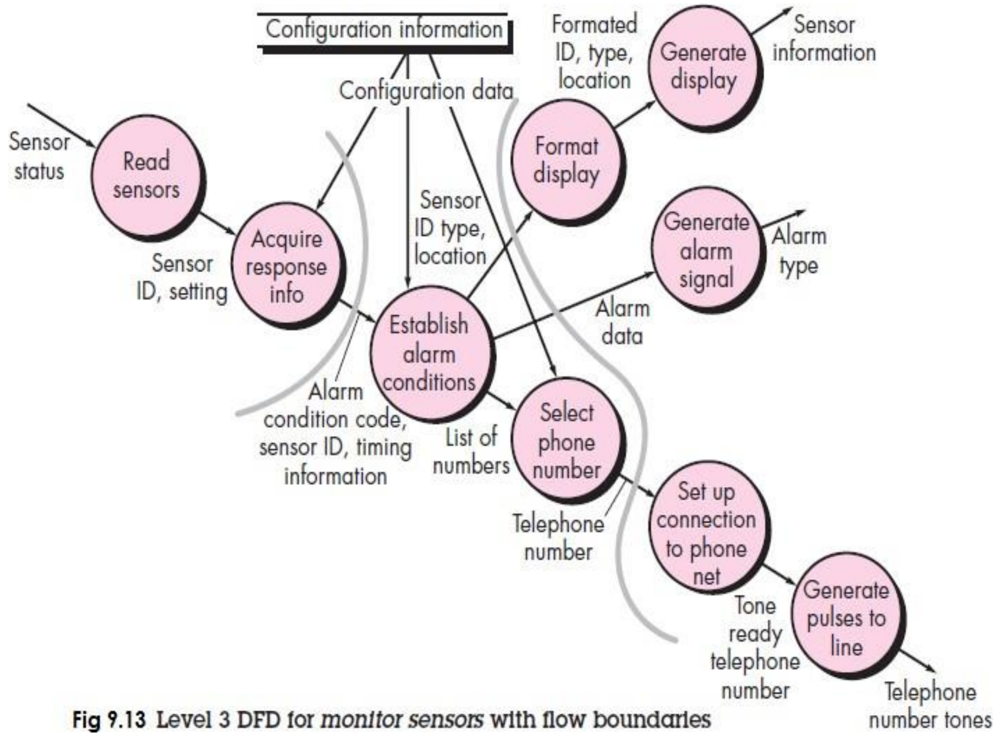


Fig 9.13 Level 3 DFD for *monitor sensors* with flow boundaries

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.

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.

➢ COMPONENT-BASED SOFTWARE ENGINEERING

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

Domain Engineering: The intent of domain engineering is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems. Domain engineering includes three major activities—analysis, construction, and dissemination.

The overall approach to domain analysis is often characterized within the context of object-oriented software engineering. The steps in the process are defined as:

Define the domain to be investigated.

Categorize the items extracted from the domain.

Collect a representative sample of applications in the domain.

Analyze each application in the sample and define analysis classes.

Develop a requirements model for the classes.

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

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.

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 othercomponents.

Security features, including access controls and authentication protocol.

Embedded design assumptions, including the use of specific numerical or non numericalalgorithms.

Exception handling

Component Adaptation. In an ideal setting, domain engineering creates a library of components that can be easily integrated into 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 post processing at the component interface to remove or mask conflicts.

Component Composition. The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure 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.

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.

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:

i. A component database capable of storing software components and the classification
ii. information necessary to retrieve them.
iii. A library management system that provides access to the database.
iv. 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.
v. CBSE tools that support the integration of reused components into a new design orimplementation.

➢ Critical systems development

A critical system is any system whose 'failure' could threaten human life, the system's environment or the existence of the organisation which operates the system.

'Failure' in this context does NOT mean failure to conform to a specification but means any potentially threatening system behaviour.

Examples of critical systems

Communication systems such as telephone switching systems, aircraft radio systems, etc.

Embedded control systems for process plants, medical devices, etc.

Command and control systems such as air-traffic control systems, disaster management systems, etc.

Financial systems such as foreign exchange transaction systems, account management systems, etc.

Critical systems usage

Most critical systems are now computer-based systems

Critical systems are becoming more widespread as society becomes more complex and more complex activities are automated

People and operational processes are very important elements of critical systems - they cannot simply be considered in terms of hardware and software

Critical systems failure

The cost of failure in a critical system is likely to exceed the cost of the system itself

As well as direct failure costs, there are indirect costs from a critical systems failure. These may be significantly greater than the direct costs

Society' s views of critical systems are not static - they are modified by each high-profile system failure

Critical systems development

Critical systems attributes are NOT independent - the systems development process must be organised so that all of them are satisfied at least to some minimum level

More rigorous (and expensive) development techniques have to be used for critical systems development because of the potential cost of failure

Developing reliable systems

Reliable systems should be 'fault-free' systems where 'fault-free' means that the system' s behaviour always conforms to its specification

Systems which are 'fault-free' may still fail because of specification or operational errors

The cost of producing reliable systems grows exponentially as reliability requirements are increased. In reality, we can never be sure that we have produced a 'fault-free' system.

Reliability achievement

Achieving systems reliability is generally based on the notion that system failures may be reduced by reducing the number of system faults

Fault reduction techniques

Fault avoidance

Fault detection

Alternatively, reliability can be achieved byensuring faults do not result in failures

Fault tolerance

➢ What is software reuse?

Software reuse is a term used for developing the software by using the existing software components. Some of the components that can be reuse are as follows;

Source code

Design and interfaces

User manuals

Software Documentation

Software requirement specifications and many more.

What are the advantages of software reuse?

Less effort: Software reuse requires less effort because many components use in the system are ready made components.

Time-saving: Re-using the ready made components is time saving for the software team.

Reduce cost: Less effort, and time saving leads to the overall cost reduction.

Increase software productivity: when you are provided with ready made components, then you can focus on the new components that are not available just like ready made components.

Utilize fewer resources: Software reuse save many sources just like effort, time, money etc.

Leads to a better quality software: Software reuse save our time and we can consume our more time on maintaining software quality and assurance.

What are Commercial-off-the-shelf and Commercial-off-the-shelf components?

Commercial-off-the-shelf is ready-made software. Commercial-off-the-shelf software components are ready-made components that can be reused for a new software.

What is reuse software engineering?

Reuse software engineering is based on guidelines and principles for reusing the existing software.

What are stages of reuse-oriented software engineering?

Requirement specification:

First of all, specify the requirements. This will help to decide that we have some existing software components for the development of software or not.

Component analysis

Helps to decide that which component can be reused where.

Reuse System design

If the requirements are changed by the customer, then still existing system designs are helpful for reuse or not.

Development

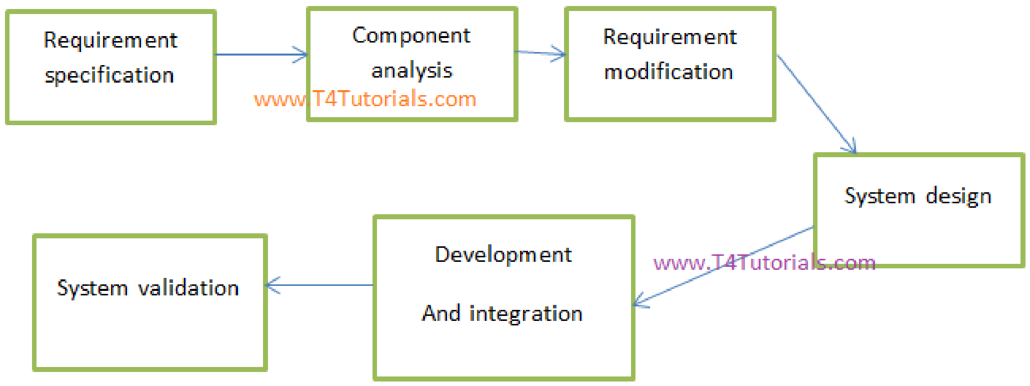
Existing components are matching with new software or not.

Integration

Can we integrate the new systems with existing components?

System validation

To validate the system that it can be accepted by the customer or not.



software reuse success factors

Capturing Domain Variations

Easing Integration

Understanding Design Context

Effective Teamwork

Managing Domain Complexity

How does inheritance promote software re-usability?

Inheritance helps in the software re-usability by using the existing components of the software to create new component.

In object oriented programming protected data members are accessible in the child and so we can say that yes inheritance promote software re-usability.

➢ USER INTERFACE DESIGN

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

Who does it? A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

Why is it important? If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won' t like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user' s perception of the software

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

What is the work product? User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

How do I ensure that I' ve done it right? An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype.

THE GOLDEN RULES

The three golden rules on interface design are

i. Place the user in control.
ii. Reduce the user' s memory load.
iii. Make the interface consistent.

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

Place the User in Control: During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface. User wanted to control the computer, not have the computer control her. Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. The result may be an interface that is easy to build, but frustrating to use. Mandel defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object is an implementation of direct manipulation.

Reduce the User's Memory Load: The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory.

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.

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

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

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

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

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction. If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de factostandard the user expects this in every application he encounters. A change will causeconfusion.

USER INTERFACE ANALYSIS AND DESIGN

Interface Analysis and Design Models: Four different models come into play when a user interface is to be analyzed and designed. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality . In addition, users can be categorized as:

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

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

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

The user's **mental model** (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response.

The implementation model combines the outward manifestation of the computer based system (look and feel interface), coupled with all supporting information (books, manuals, videotapes, help) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "Know the user, know the tasks."

The Process: The analysis and design process for user interfaces is iterative and can be represented using a spiral model. Referring to Figure 11.1, the four distinct framework activities:

(1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure 11.1 implies that each of these tasks will

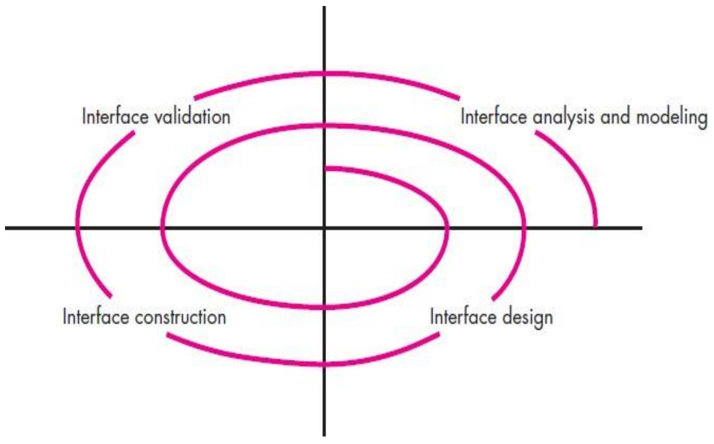


Fig 11.1: The user interface design process

occur more than once, with each pass around the spiral representing additional elaboration ofrequirements and the resultant design.

In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed. Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified,described, and elaborated.

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

Where will the interface be located physically?

Will the user be sitting, standing, or performing other tasks unrelated to the interface?

Does the interface hardware accommodate space, light, or noise constraints?

Are there special human factors considerations driven by environmental factors?

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

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

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

Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

INTERFACE ANALYSIS

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

User Analysis: The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use thesystem. Information from a broad array of sources can be used to accomplish this:

User Interviews. The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn' t, what users like and what they dislike, what featuresgenerate questions and what features are easy to use.

The following set of questions will help you to better understand the users of a system:

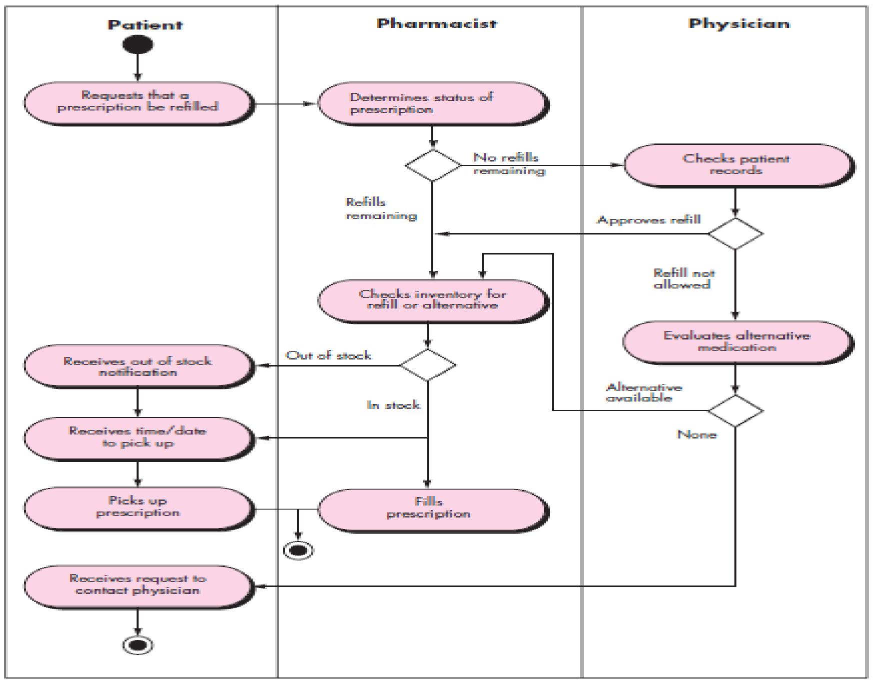
Are users trained professionals, technicians, clerical, or manufacturing workers?

What level of formal education does the average user have?

Are the users capable of learning from written materials or have they expressed a desire forclassroom training?

Are users expert typists or keyboard phobic?

What is the age range of the user community?
Will the users be represented predominately by one gender?
How are users compensated for the work they perform?
Do users work normal office hours or do they work until the job is done?
Is the software to be an integral part of the work users do or will it be used only occasionally?
What is the primary spoken language among users?
What are the consequences if a user makes a mistake using the system?
Are users experts in the subject matter that is addressed by the system?
Do users want to know about the technology that sits behind the interface?
completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swim lane diagram (a variation on the activity diagram). See Figure 11.2


Patient
Pharmacist
Physician
Requests that a prescription be refilled
Determines status of prescription
No refills remaining
Checks patient records
Refills remaining
Approves refill
Refill not allowed
Checks inventory for refill or alternative
Evaluates alternative medication
Receives out of stock notification
Out of stock
In stock
Alternative available
None
Receives time/date to pick up
Picks up prescription
Fills prescription
Receives request to contact physician
Fig 11.2: Swimlane diagram for prescription refill function

Regardless, the flow of events (shown in the figure) enables you to recognize a number of keyinterface characteristics:

Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.

The interface design for pharmacists and physicians must accommodate access to

and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
Many of the activities noted in the swim lane diagram can be further elaborated using task analysis and/or object elaboration (e.g., Fills prescription could imply a mail-order delivery, a visitto a pharmacy, or a visit to a special drug distribution center).
Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the following user task and subtask hierarchy.
User task: Requests that a prescription be refilled
Provide identifying information.
Specify name.
Specify userid.
Specify PIN and password.
Specify prescription number.
Specify date refill is required.
To complete the task, three subtasks are defined. One of these subtasks, provide identifyinginformation, is further elaborated in three additional sub-subtasks.
Analysis of Display Content: The user tasks lead to the presentation of a variety of different types of content. For modern applications, display content can range from character- based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a pictureof a person), or specialized information (e.g., audio or video files).
These data objects may be
(1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmittedfrom systems external to the application in question.
During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

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

INTERFACE DESIGN STEPS

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design is an iterative process. Although many different user interface design models (e.g.,have been proposed, all suggest some combination of the following steps:
Using information developed during interface analysis define interface objects and actions(operations).
Define events (user actions) that will cause the state of the user interface to change. Modelthis behavior.
Depict each interface state as it will actually look to the end user.
Indicate how the user interprets the state of the system from information provided through theinterface.
Applying Interface Design Steps: The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted.

User Interface Design Patterns: Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. A vast array of interface design patterns has been proposed over the past decade.

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

Response time. System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. Variability refers to the deviation from average response time.

Help facilities. Almost every user of an interactive, computer-based system requires help now and then. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface. A number of design issues must be addressed when a help facility is considered:

Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.

How will the user request help? Options include a help menu, a special function key, or a HELP command.

How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.

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

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

The message should describe the problem in jargon that the user can understand.

The message should provide constructive advice for recovering from the error.

The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."

The message should be "nonjudgmental." That is, the wording should never place blame on the user.

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

Application accessibility. As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. Accessibility for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines - many designed for Web applications but often applicable to all types of software— provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

➢ Web Apps Design Issues

In plain terms, an application is a software program that runs on a device. A web app is one that' s stored on the Internet and can be accessed from any browser. They aren' t constrained to any one operating system, don' t take up hard drive space on your device, and can be used by anyone with an Internet connection.

Web applications can include everything from productivity-enhancing collaboration tools (e.g. Google Drive and Slack) to less productive (but often fun) games (e.g. Candy Crush and Words with Friends). Popular web apps are also often available as mobile apps too.

Web application development challenges during planning

1. Clearly defining your goals.

Clearly defining your goals and requirements will make or break your web app. Many of the other challenges that we' ll talk about later, such as application performance and speed, can be almost entirely addressed by making thoughtful choices during the planning stage.

It all starts with your vision for your app. That affects everything that follows. The key requirements for an enterprise-level collaborative app are different than the requirements for a fitness app or a game.

There are a few key considerations in any development process:

- Who are your intended users?
- What experience do you want to give them?
- What are your must-have design features?
- What are your technical requirements?

Once you have defined all of these things, you can start to address them

2. Choosing the right tech stack.

Throughout this section we’ re going to talk about tech stacks. They’ re a combination of programming languages, frameworks, servers, software and other tools used by developers. In essence, they’ re the set of technologies a web development and design agency uses to build a web or mobile app development.

You should always choose a tech stack that aligns with the problems you are trying to solve. For example, you probably won’ t need a complex tech stack for a simple web application, or a tech stack that helps you optimize for scalability when your user base will be a consistent size.

You should consider whether your tech stack is widely used in the industry. With industry-standard tech stacks, you will have a large pool of skilled developers to draw from for your initial build and future requirements.

Make sure to choose a tech stack that has thorough documentation. It’ s almost inevitable that you’ ll run into a troubleshooting problem at some point during development. Great documentation and support can save you time, money and hassle.

3. UX

User experience (UX) encapsulates the reactions, perceptions, and feelings your users experience while engaged in your application. It’ s the feeling of ease and simplicity that you get from great design. It’ s also the frustration that you feel when interacting with poor design.

That’ s why it’ s important to think about the overall impression you want to leave on your users before you start making detailed decisions about how to build it.

If you deliver effectively to the deeper needs of all your users – providing pleasure, engagement, and an overall emotional appeal – your application can become a central part of your users’ day-to-day lives and deliver to your business goals.

4. UI & simplistic design

User Interface (UI) design includes all the visual elements your users interact with on your web application. It is everything that your users see on their screens and everything they click on to guide them through the experience.

Great UI design certainly makes your application visually appealing, but it goes beyond just simple aesthetics. The goal is to make the actual user experience simple and accessible – and usable. This means using only a targeted, purposeful selection of copy and content, making clear the options your users have throughout the experience and ensuring information is readily available at each step.

Intuitive UI typically involves:

- Clear navigation
- Engaging visuals
- Easy-to-read typography

Here's a succinct way to think about the relationship between UX design and UI design: great UI makes it easy for your users while great UX makes it meaningful.

5. Performance and speed

No user likes slow load times. And they can have real consequences for your business. If your application is slow, users won't wait. They'll leave. This is the reality of web application development issues today. You may only get a single chance to hook a user on your product.

So, if you know that you're building an application with a lot of content (e.g. videos), you should outline that upfront so your developers can build a more robust application to ensure performance. Likewise, be clear about any intentions you have to scale your application rapidly. You don't want your application to slow down if it makes a splash in the market or you see periodic surges in traffic.

This kind of planning for the future will ensure your initial app launch can deliver the kind of speed and performance you want for your early, often-critical, users. It will also mean that your initial build will provide the foundation you need for future business growth.

6. Scalability The challenge of scalability relates to how you want your application to develop over time. If you want your application built right today, you'll need to know as much as possible about what you need it to do in the future.

Your application might include fairly lean content at launch but, a year or two in, you could be planning a detailed, expansive content-rich experience. This is where extensibility comes in. Extensibility is when an application is initially designed to incorporate new capabilities and functionality in the future. This can be integrated right from the start of the development process. If you can articulate a long-term vision for your app, it can be built to grow and evolve over time.

7. Web security threats

There are a number of things to prioritize so your application and your users are secure. Choosing the right development infrastructure is one. Make sure that the infrastructure you build on has enough security services and options so your developers can implement the proper security measures for your application.

SSL certificates are a global standard security technology that enables encrypted communication between your web browser and server. When integrated in your app, they enhance its security and eliminate the chance it is flagged as unsecure by web browsers. SSL certificates also help protect credit-card numbers in ecommerce transactions and other sensitive user information like usernames, passwords and email addresses. In essence, SSL allows for a private "conversation" just between the two intended parties and hides sensitive information from hackers and identity thieves.

Creating robust password requirements and multi-factor authentication for your users are also effective security measures. More complex passwords are less likely to be hacked. Multi-factor authentication, where your users take multiple steps to confirm their identities, also gives your application an added layer of protection.

Conclusion

- Planning strategically, and in detail, will mitigate your toughest development challenges. This is where you should always start. Set business goals. Then ensure everything that follows is in service to them.
- Always remember that your web application will feature a blend of design elements and build (technical) elements. It needs to deliver to both if you want to meet your business goals. You can' t overcome bad design with a good build, or the reverse.
- Your application is for your users. Remember you are serving your own business goals by serving your users first, last and always.

➢ What Is Web Application Architecture?

Web app architecture is a kind of software architecture that describes the processes associated with programs running in a browser. That' s why the architecture of mobile apps or programs for IoT uses another type of digital architecture.

Web app architecture contains a set of components and a description of their logic interaction. Within the web development process in general, it determines the future design of your product, its IT infrastructure, user experience, software modules (web architecture design), as well as the promotion and monetization of your future web application. Working on it is the first step when creating your product.

Based on decisions related to the high-level architecture of an app (for example, whether it will be monolithic or based on the micro service pattern), the team forms more specific technical requirements for the future software and its tiers and plans further work.

Here are the following parameters by which the quality of web app architecture can be described:

Safety level and stability

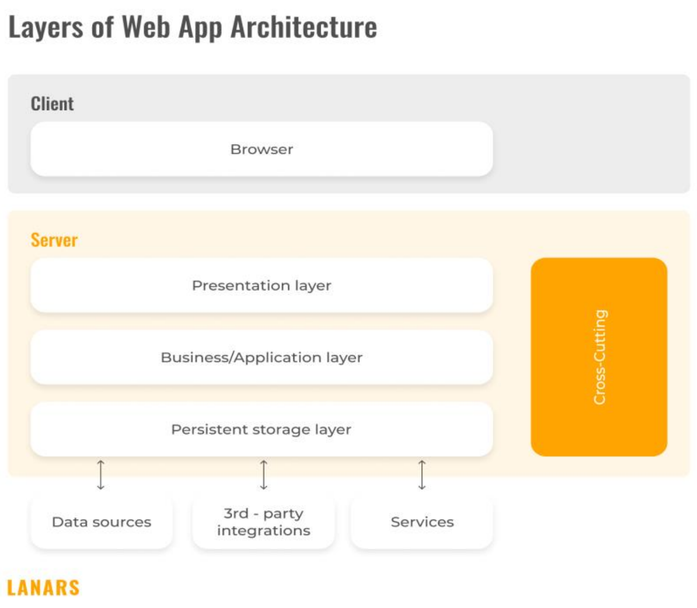
- Request processing speed
- Component reusability
- Clarity towards program code
- Ability to collect analytics and test different components independently
- Scalability of the product and its components
- System automation level

Why Does Web Architecture Matter

As we already know, web application architecture design is the first stage of software development, and web architecture, in turn, is the backbone of your web app. So what makes digital product architecture so important?

First, the web architecture is literally the foundation onto which you put all the other product components. If this foundation is solid and stable, further work on the product will be time- and cost-effective. If you make mistakes at this critical stage, all the other stages of software development, including scale changes or writing components using a particular programming language, will be slowed down.

Second, high-level web architecture is difficult to amend or modify in the later stages of product development. Typically, major changes of this kind literally mean you have to rebuild the product you want from scratch. Such a decision means postponing the release date, as well as losing financial resources associated with your inability to use components from the previous version of your Node.js, Python, or Java web application architecture.



Third, web app maintenance can become more expensive if you made bad decisions at the overall architecture development stage.

Web Application Architecture Diagram

A person uses a browser to enter their request (this can be an address of the website they want to go to (URL), or a command they send via a web page interface).

The browser translates the user's request into digital language, and then determines the path to the site that the user needs and requests access to it.

1. When the server is found, it receives the browser request, processes it, and, in response to it, sends the required data.
2. The user's browser processes this data and displays the result on the user's screen: a new or updated web page.

Modern web architecture is based on the principles described in this diagram. While this scheme of user-server interaction may seem simple and straightforward, there are many approaches and architectural patterns that are suitable for specific types of web applications.

Components of Web Application Architecture

App components are the most important part of web application architecture, but not the only part. As well as this, you will be dealing with data for your database and middleware system design. Nevertheless, it is the components of your system that determine the functions of your future product and the quality of the user experience that your web application will offer to Internet users.

Modern web application architecture operates in two categories of components: structural and UI & UX components. Components of the first group are created using programming languages such as Java, .Net, NodeJS, or Python. UI and UX components are created by designers. After being created, their design layouts are commonly sent for further implementation by programmers in the form of a working web application interface.

Structural (server) components: your server and database

UI and UX (client) components: includes interface notification elements and input controls, navigational components, admin and other dashboards, design layouts, activity tracking tools, informational elements, and many others

The goal of the UI and UX web application architecture components is to design a perfect user experience. At the same time, structural components are more about the smooth running of the web app and its features. You can use a web application model that consists of one server and one database, or you can design an application that operates on multiple servers and databases. The second approach is considered to be more sustainable and reliable.