UNIT III

DESIGN ENGINEERING

Syllabus

Design Engineering: Design process and design quality, design concepts, the design model. **Creating an architectural design:** software architecture, data design, architectural styles and patterns, architectural design, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams.

***

Design Engineering:

Design:

Design is a meaningful engineering representation of something that is to be built.

It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design.

In the software engineering context, design focuses on four major areas of concern: ***data, architecture, interfaces, and components.***

Software engineers design computer based systems, but the skills required at each level of design work are different.

- At the data and architectural level, design focuses on patterns as they apply to the application to be built.
- At the interface level, human ergonomics (human factors) often dictate our design approach.
- At the component level, a "programming approach" leads us to effective data and procedural designs.

Why is it important?

Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers.

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, you must practice diversification and then convergence.

Design Engineering:

Design engineering encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.

◎Design principles establish an overriding philosophy that guides you in the design work you must perform.

◎Design concepts must be understood before the mechanics of design practice are applied.

◎Design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design Within the Context of Software Engineering:

"The most common miracle of software engineering is the transition from analysis to design and design to code."

Once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in figure below.

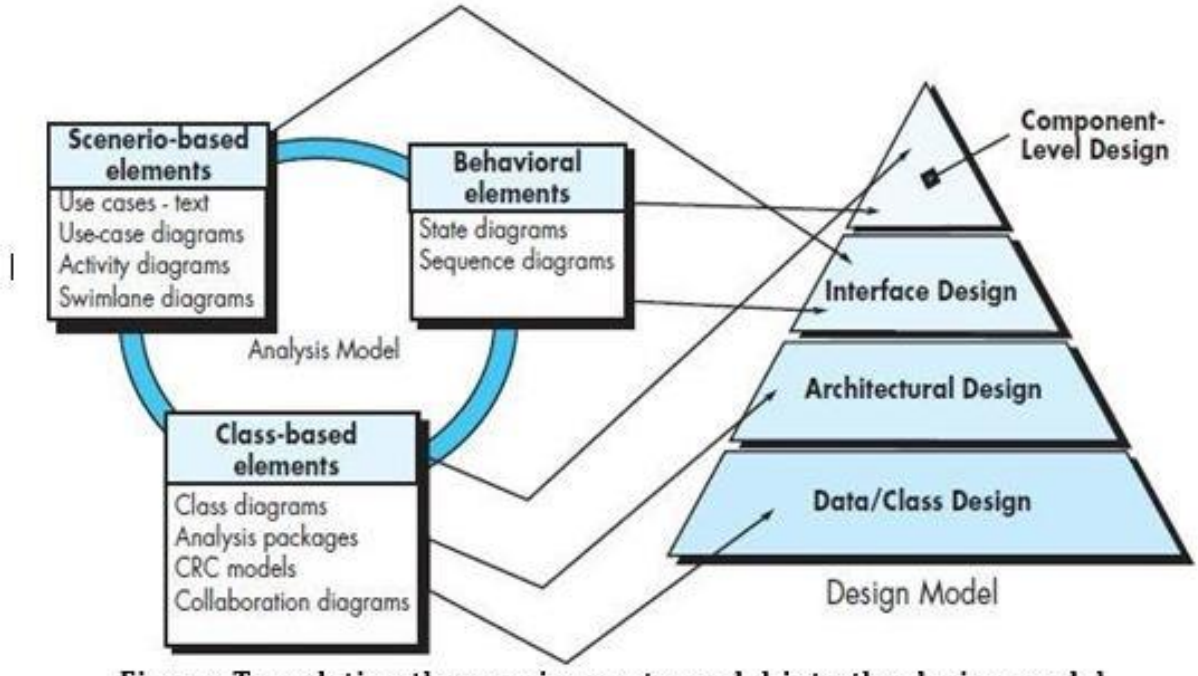


Figure: Translating the requirements model into the design model

The requirements model, manifested by scenario-based, class-based, flow- oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.
- Part of class design may occur in conjunction with the design of software architecture.
- More detailed class design occurs as each software component is designed.
- The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.
- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.
- 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.

oOo

Design Process and Design Quality:

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

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs. McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

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

Each of these characteristics is actually a goal of the design process.

Quality Guidelines:

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

The following are the quality guidelines:

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

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

Quality Attributes:

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability.

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

- **Functionality** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- **Usability** is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- **Reliability** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.
- **Performance** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability** combines the ability to extend the program (extensibility), adaptability, serviceability - these three attributes represent a more common term, maintainability and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed.

- One application may stress functionality with a special emphasis on security.
- Another may demand performance with particular emphasis on processing speed.
- A third might focus on reliability.

Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

Design Concepts:

A set of fundamental software design concepts has evolved over the history of software engineering.

Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

M. A. Jackson once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide the necessary framework for "getting it right."

Following are the important software design concepts that span both traditional and object-oriented software development.

Abstraction:

Each step in the software process is a refinement in the level of abstraction of the software solution.

Many levels of abstraction are there.

- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.
- As we move through different levels of abstraction, we work to create procedural and data abstractions.
- A procedural abstraction is a named sequence of instructions that has a specific and limited function.

An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

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

In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

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. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

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

- **Structural models** represent architecture as an organized collection of program components.
- **Framework models** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.
- **Dynamic models** address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- **Process models** focus on the design of the business or technical process that the system must accommodate.
- Finally, **functional models** can be used to represent the functional hierarchy of a system.

Patterns:

Brad Appleton defines a design pattern in the following manner: "a pattern is a named nugget of inside which conveys that essence of a proven solution to a recurring problem within a certain context amidst competing concerns."

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

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

- Whether the pattern is capable to the current work,
- Whether the pattern can be reused,
- Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Modularity:

Software architecture and design patterns embody modularity; 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". Monolithic software cannot be easily grasped by a

software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

The "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 modularity and software. If we subdivide software indefinitely, the effort required to develop it will become negligibly small. The effort 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 associated with integrating the modules also grow.

Under modularity or over modularity should be avoided. We modularize a design so that development can be more easily planned; software increment can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Information Hiding:

The principle of information hiding suggests that modules be "characterized by design decision that hides from all others."

Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

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 entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and local data structure used by 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 procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within software.

Functional Independence:

The concept of functional independence is a direct outgrowth of 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, we want to design software so that each module addresses a specific sub function of requirements and has a simple interface when viewed from other parts of the program structure.

Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified. Independent sign or code modifications are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

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

- Cohesion is an indication of the relative functional strength of a module.
- Coupling is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information hiding.

A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing.

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, we strive for 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 originally proposed by Niklaus Wirth.

A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function in a step wise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

Refactoring :

Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior.

Fowler defines refactoring in the following manner: "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code(design) yet improves its internal structure."

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 designer may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Design classes:

The software team must define a set of design classes that

- Refine the analysis classes by providing design detail that will enable the classes to be implemented, and
- Create a new set of design classes that implement a software infrastructure to support the design solution.

Five different types of design classes, each representing a different layer of the design architecture are suggested.

User interface classes: define all abstractions that are necessary for human computer interaction. In many cases, HCL occurs within the context of a metaphor 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 defined earlier. The classes identify the attributes and services 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 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.

As the design model evolves, the software team must develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation.

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

- **Complete and sufficient:** A design class should be the complete encapsulation of all

attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

- **Primitiveness:** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.
- **High cohesion:** A cohesive design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.
- **Low coupling:** Within the design model, it is necessary for design classes to collaborate with one another. 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 classes in other subsystems. This restriction, called the law of Demeter, suggests that a method should only sent messages to methods in neighboring classes.

The Design Model:

The design model can be viewed in two different dimensions.

- The process dimension indicates the evolution of the design model as design tasks are executed as a 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.

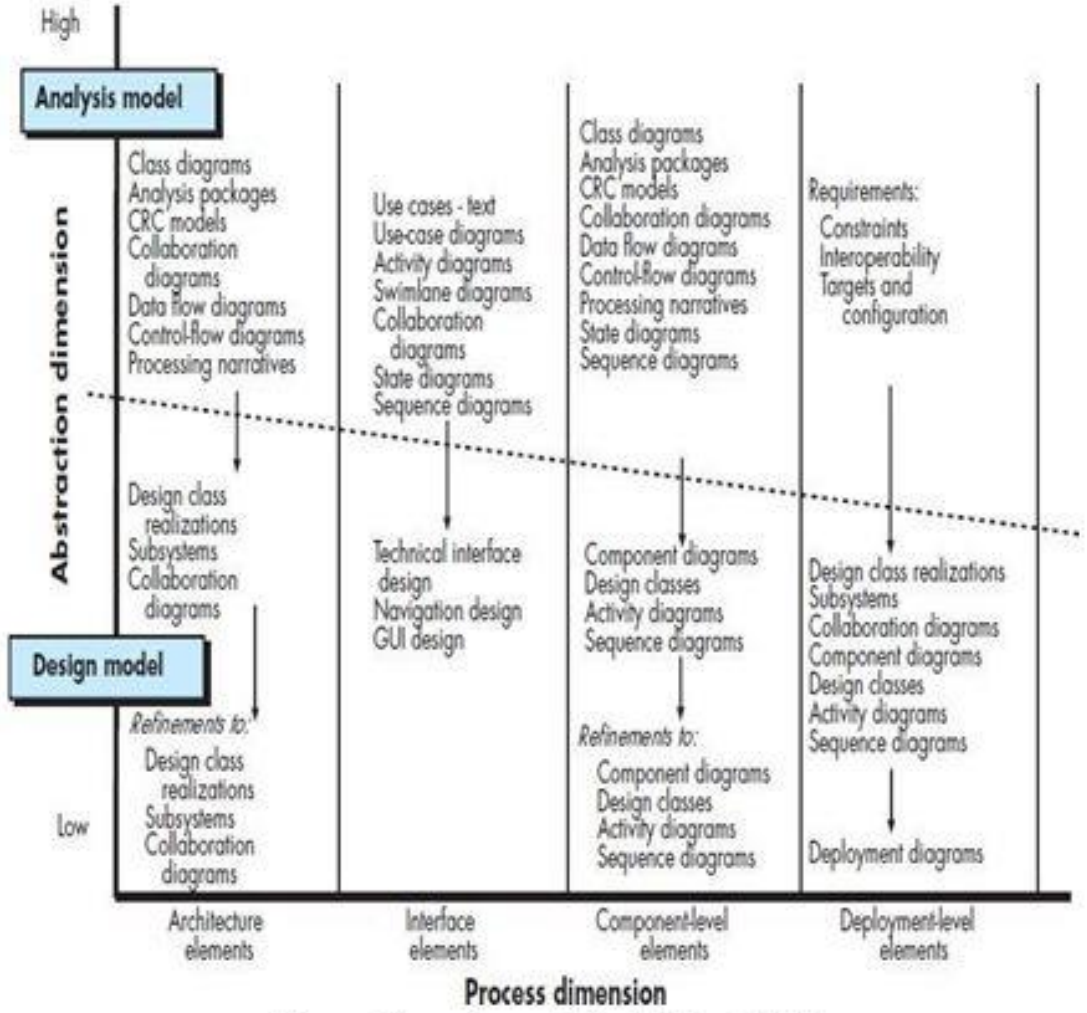


Figure: Dimensions of the Design Model

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

It is important to mention however, that model elements noted along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model us usually delayed until the design has been fully developed.

Data design elements:

Data design sometimes referred to as data architecting creates a model of data and/or information that is represented at a high level of abstraction. 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 criterion 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 architectural model is derived from three sources.

- Information about the application domain for the software to be built.
- Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
- The availability of architectural patterns

Interface design elements:

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

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

There are 3 important elements of interface design:

- The user interface (UI);
- External interfaces to other systems, devices, networks, or other produces or consumers of information; and
- Internal interfaces between various design components.

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

UI design is a major software engineering action.

The design of a UI 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). In general, the UI is a unique subsystem within the overall

application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of external interfaces should incorporate error checking and appropriated security features.

UML defines an interface in the following manner:" an interface is a specifier for the externally- visible operations of a class, component, or other classifier without specification of internal structure."

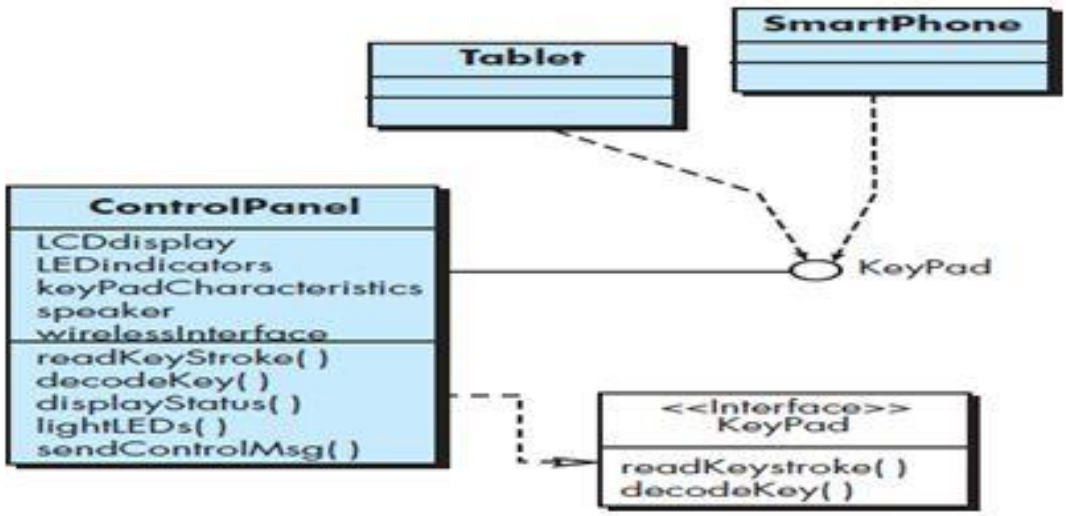


Figure: UML Interface representation of ControlPanel

COMPONENT-LEVEL DESIGN ELEMENTS

The component-level design for software is equivalent to a set of detailed drawings.

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.

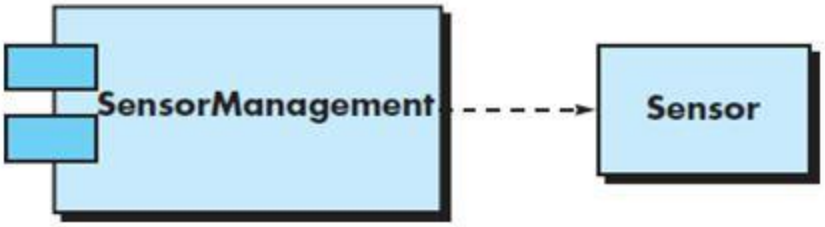


Figure: UML Component Diagram

Deployment-level design elements:

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

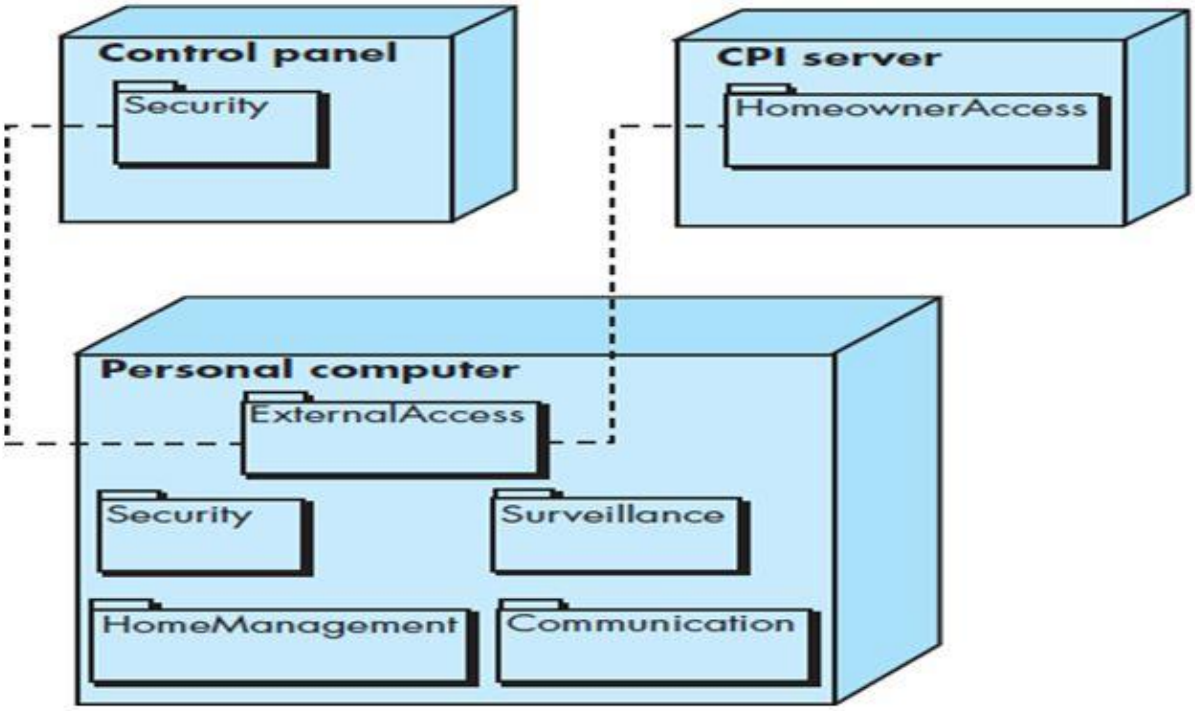


Figure: UML Deployment Diagram

Creating an Architectural Design

The architectural design is the preliminary blueprint from which software is constructed.

Software Architecture :

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

➢ What is Architecture?

"The architecture of a system is a comprehensive framework that describes its form and structure - its components and how they fit together."

In simple words, architecture captures system structure in terms of components and how they interact.

Architecture is defined as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

➢ What is Software Architecture?

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

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

1. Analyze the effectiveness of the design in meeting its stated requirements,
2. Consider architectural alternatives at a stage when making design changes is still relatively easy, and
3. Reduce the risks associated with the construction of the software.

➢ Why Is Architecture Important?

There are three key reasons stating that software architecture is important:

1. Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
2. 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.

3. Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

Design of Software Architecture :

The design of software architecture considers two levels of the design pyramid:

1. Data Design
2. Architectural Design

Data design enables us to represent the data components of the architecture in conventional systems and class definitions (encapsulating attributes and operations) in object-oriented systems.

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

➢ Data Design :

The data design action translates data objects defined as a part of the analysis model into data structures at the software component level and whenever necessary, a database architecture at the application level.

➢ Data Design at the Architectural Level :

Today, businesses have dozens of databases serving many applications encompassing hundreds of gigabytes of data. The Challenge is to extract useful information from this data environment, particularly when the information desiredis cross-functional (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

To solve this challenge, the business IT community has developed data mining techniques, also called *knowledge discovery in databases (KDD)*, that navigate through existing databases in an attempt to extract appropriate business- level information. However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment.

An alternative solution, called a data warehouse, adds an additional layer to the data architecture. A data warehouse is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business. In a sense, a data warehouse is a large, independent database that has access to the data that are stored in databases that serve the set of

applications required by a business.

➢ Data Design at the Component Level :

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. Wasserman has proposed a set of principles that may be used to specify and design such data structures. Following are the set of principles for data specification :

1. **The systematic analysis principles applied to function and behavior should also be applied to data.** Representations of dat flow and content should also be developed and reviewed. Data objects should be identified, alternative data organizations should be considered, and the impact of data modeling on software design should be evaluated.
2. **All data structues and the operations to be performed on each should be identified.** The design of an efficient data structure must take the operations to be performed on the data structure into account. The attributes and operations encapsulated within a class satisfy this principle.
3. **A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it.** Class diagrams define the data items(attributes) contained within a class and the processing(operations) that are applied to these data items.
4. **Low-level data design decisions should be deferred until late in the design process.** A process of stepwise refinement may be used for the design of data. That is, overall data organization may be defined during requirements analysis, refined during data design work, and specified in detail during component-level design.

   The represenattion of a data structure shold be known only to those modules that must make direct use of the data contained within the **structure.** The concept of information hiding and the related concept of coupling provide important insight into the quality of a software design.
5. **A library of useful data structures and the operations that may be applied to them should be developed.** A class library achieves this.
6. **A software design and programming language should support the specification and realization of abstract data types.** The implementation of sophisticated data structure can be made exceedingly difficult if no means for direct specification of the structure exists in the programming language

chosen for implementation.

These priciples form a basis for a component-level data design approach that canbe integrated into both the analysis and design activities.

Architecture Styles and Patterns:

The architectural style is also a template for construction. The software thatis built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

1. A set of computers that perform a function required by a system.
2. A set of connectors that enable "communication, coordination, andcooperation" among components
3. Constraints that define how components can be integrated to form the system
4. Semantic models that enable a designer to understand the overall propertiesof a system by analyzing the known properties of its constituent parts.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways:

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

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

(3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

A Brief Taxonomy of Architectural Styles:

- **Data-centered architecture:**

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, orotherwise modify data within the store. Following figure illustrates a typical data-centered style.

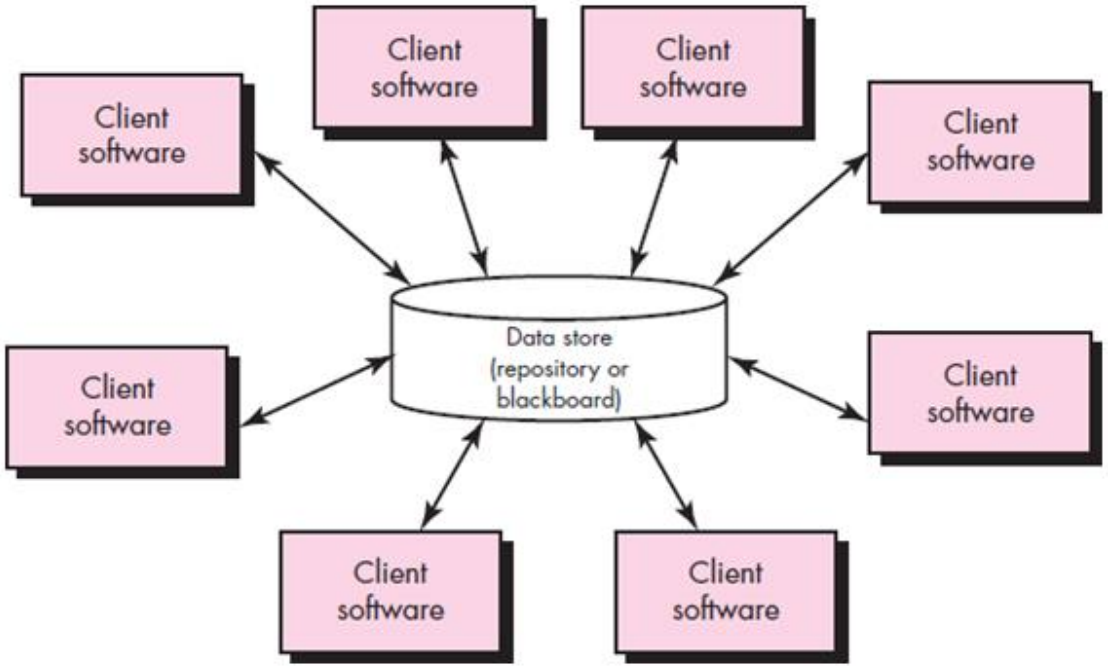


Figure: Data-centered Architecture

Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation onthis approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote *integrability*.

- Data-flow architecture:

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independentlyof 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 ofits neighboring filters. If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

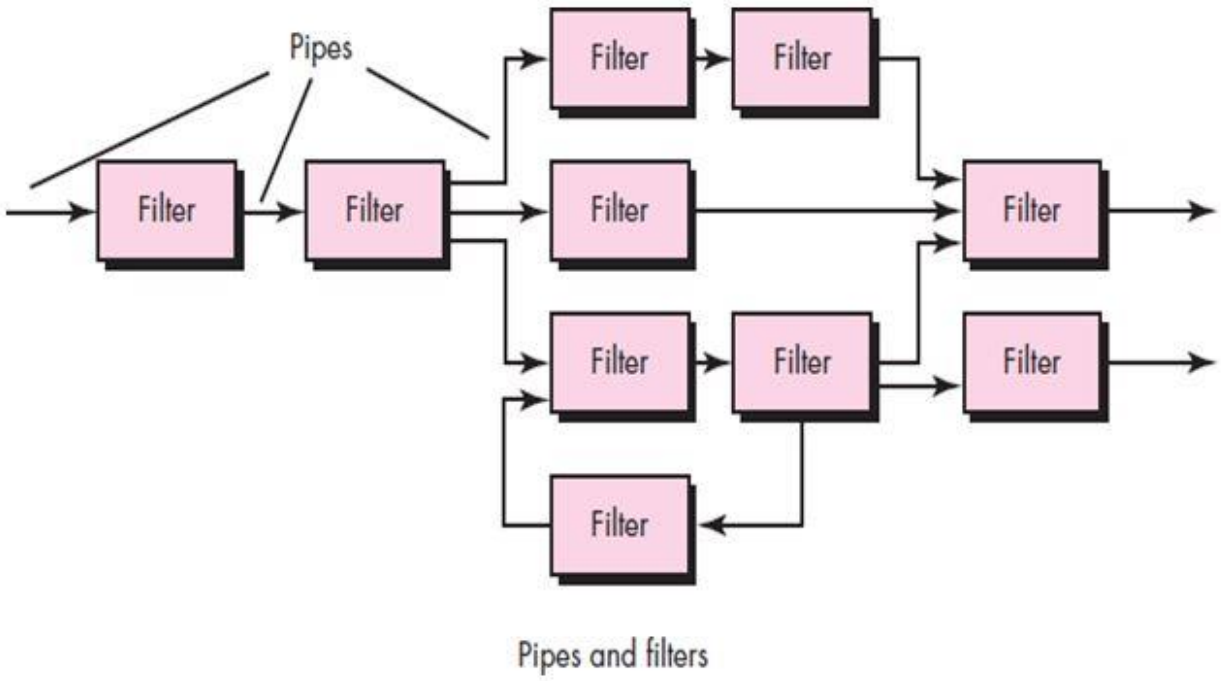


Figure: Data-flow Architecture

- **Call and return architecture:**

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:

1. ***Main program/subprogram architecture:*** 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. Following figure illustrates architecture of this type.

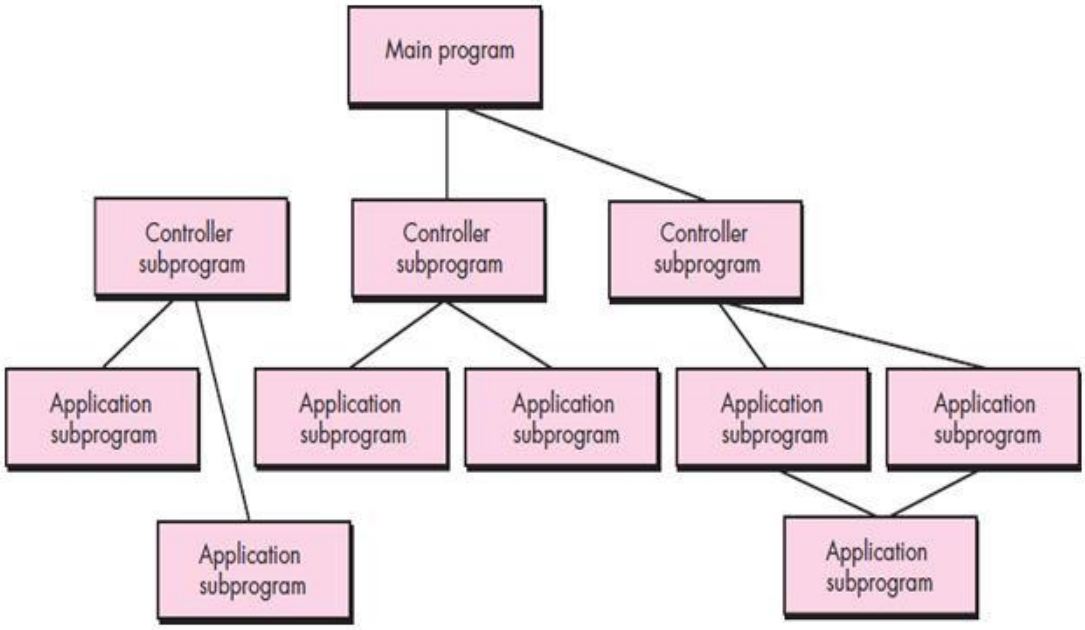


Figure: Main program/Sub program Architecture

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

➢ Object-oriented architecture:

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 architecture:

The basic structure of a layered architecture is illustrated in the following figure

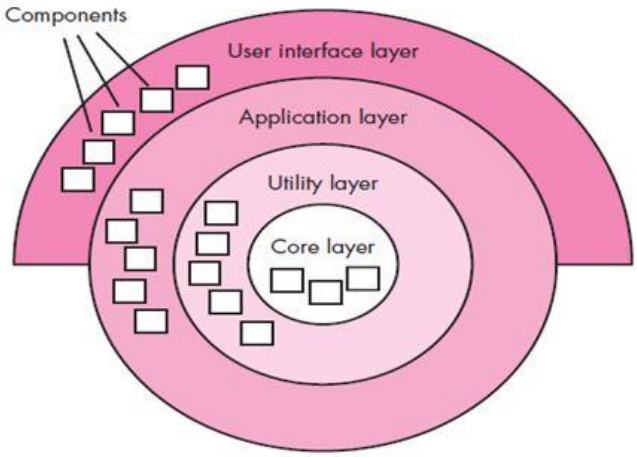


Figure: Layered Arhitecture

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.

Architectural Patterns:

A software architecture may have a number of architectural patterns thataddress issues such as concurrency, persistence and distribution.

- **Concurrency:** Many applications must handle multiple tasks in a manner thatsimulates parallelism.

  For example:

  - ✓ *operating system process management* pattern
  - ✓ *task scheduler* pattern
- **Persistence:** Data persists if it survives past the execution of the process thatcreated it.

  For example:
  - ✓ a ***database management system*** pattern that applies the storage andretrieval capability of a DBMS to the application architecture
  - ✓ an ***application level persistence*** pattern that builds persistence features into the application architecture
- **Distribution:** The manner in which systems or components within systemscommunicate with one another in a distributed environment

For example: A ***broker*** acts as a 'middle-man' between the client componentand a server component. CORBA is an example of a broker architecture.

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 ofcomponents 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 systemsporadically?
- ✓ What is the mode of data transfer (i.e., are data passed from one componentto 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:

As architectural design begins, the software to be developed must be put into context – that is, the design should define the external entities(other systesms, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the analysis model and all other information gathered during requirements engineering.

Once context is modeled and all external software interfaces have been described, the designer specifies the structure of the system by defining and refining software components that implement the architecture.

Thies process continues iteratively until a complete architectural structure has been derived.

Representing the system in context:

At the architectural design level, a software architect uses an architectural context diagram(ACO) 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.

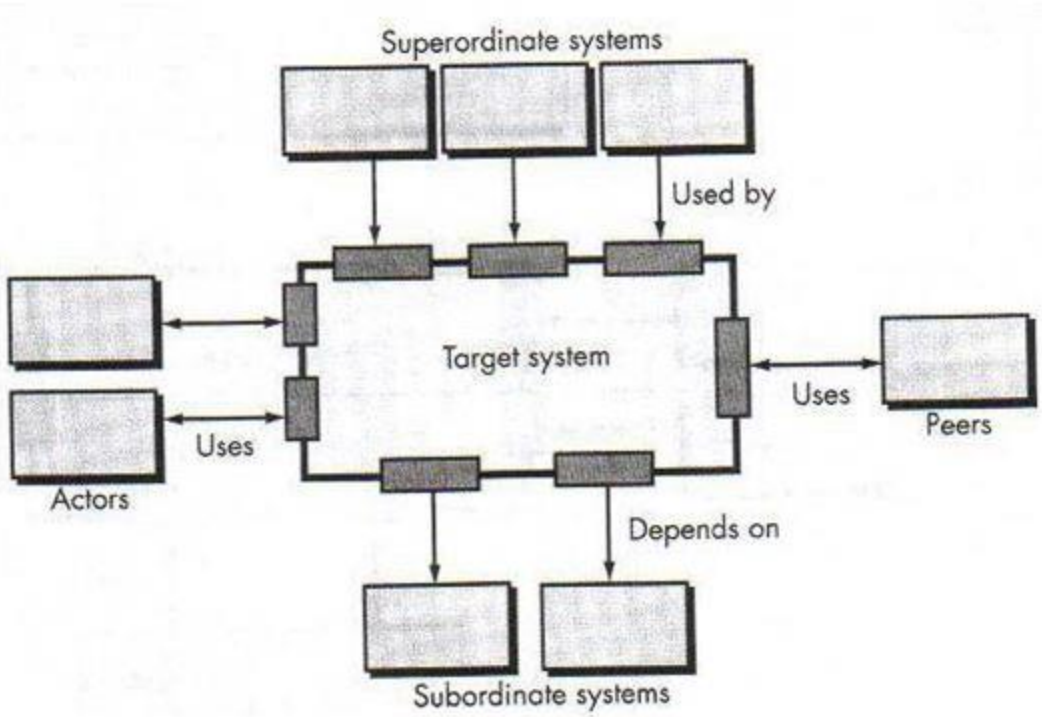


Figure: Architectural Context Diagram

Systems that interacts with the target system are represented as:

Superordinate systems: These are the systems which consider the (use the) target system inorder to complete its higher valued activities.

Subordinate systems: These are the systems which funtion along with the target system. Hence, supporting the target system in successfully completing its processing.

Peer systems or Peers: These are the systems which directly interact with the target system same as client-server interaction.

Actors: These are specimens or any entities possesing a definite set of roles and interacting with the system. During this interaction an actor can either provide or accept information from the system.

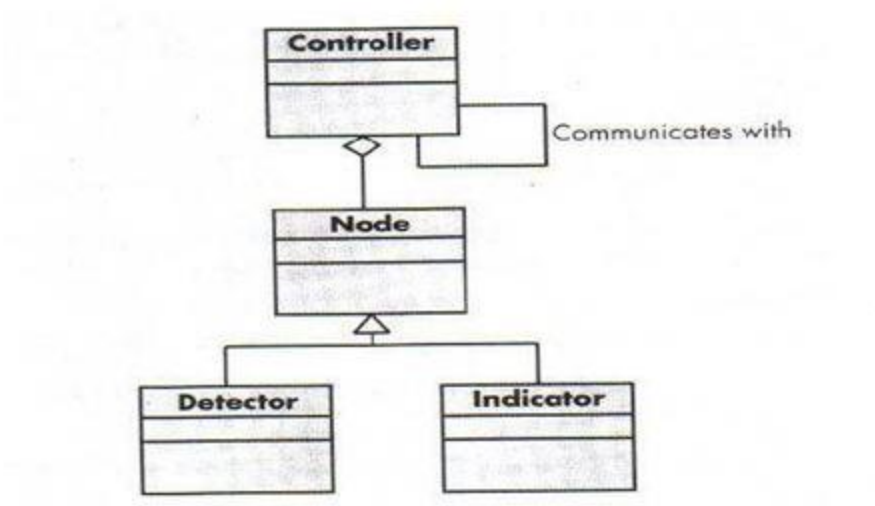
Each of these external entities communicates with the target system through an interface(the small shaded rectangles)

Following is the ACD(Architectural Context Diagram) depicting the safe home security systems.

Indicator – An abstraction that represents all mechanims for indicating that an alarm condition is occuring.(e.g. alarm siren, flash lights, bell).

Controller- An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controller reside on a network, they have the ability to communicate with one another.

Each of these archetypes is depicted using UML notation as shown in figure:



UML Relationships for Safe Home Security function archetypes

As archetypes represents only abstractions. Hence, they can be further refined into components just by refining these abstractions. For example, detector might be refined into a class hierarchy of sensors.

Refining the Architecture into components:

As the software architecture is refined into components the structure of the system begins to emerge, for this purpose we initially consider the classes which were described as part of the analysis mode. These analysis classes forms the major entities of application domain. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate components but

have no business connection to the application domain.

The interfaces depicted in the architecture context diagram imply one or more specialized components that process the data that fow across the interface. In some case graphical user interface, a complete subsystem architecture with many components must be designed.

- Components of the software architecture are dervied from three sources:
  - The application domain
  - The infrastructure domain
  - The interface domain

For safe home security system, we might define the set of top-level components as follows:

External Communication Management- Coordinates communication of the security function with external entities, for example, internet-based systesm, external alarm notification.

Control Panel Processing-manages all control panel functionality.

Detector Management- Coordinates access to all detectors attached to the system

Alarm Processing- verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall SafeHome architecture. Design classes(with appropriate attributes and operations) would be defined for each. It is important to note, how-ever, that the design details of all attributes and operations would not be specified until component-level design.

The overall architectural structure is illustrated in the following figure.

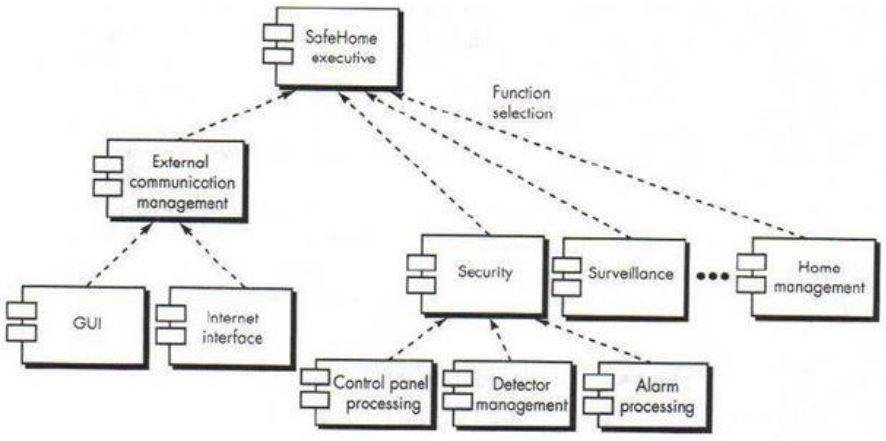


Figure: Overall architectural structure for Safe Home with top-level components

Transactions are acquired by external communication management as they move in from components that process the SafeHome GUI and the internet interface. This information is managed by a SafeHome executive componet that selects the appropriate product function. The control panel processing component interacts with homeowner to arm/disarm the security fucntion. The dector management component polls sensors to detect an alaram condition, and ahte alarm processing component produces output when alarm is detected.

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 (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this I mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

The following figure illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in overall architecture are elaborated toshow additional detail. For example, the *detector management* component interacts with a *scheduler* infra-structure component that implements polling of each *sensor* object used by the security system.

Assessing Alternative Architectural Designs:

At its best, design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. In the sections that follow, we consider the assessment of alternatives architectural designs.

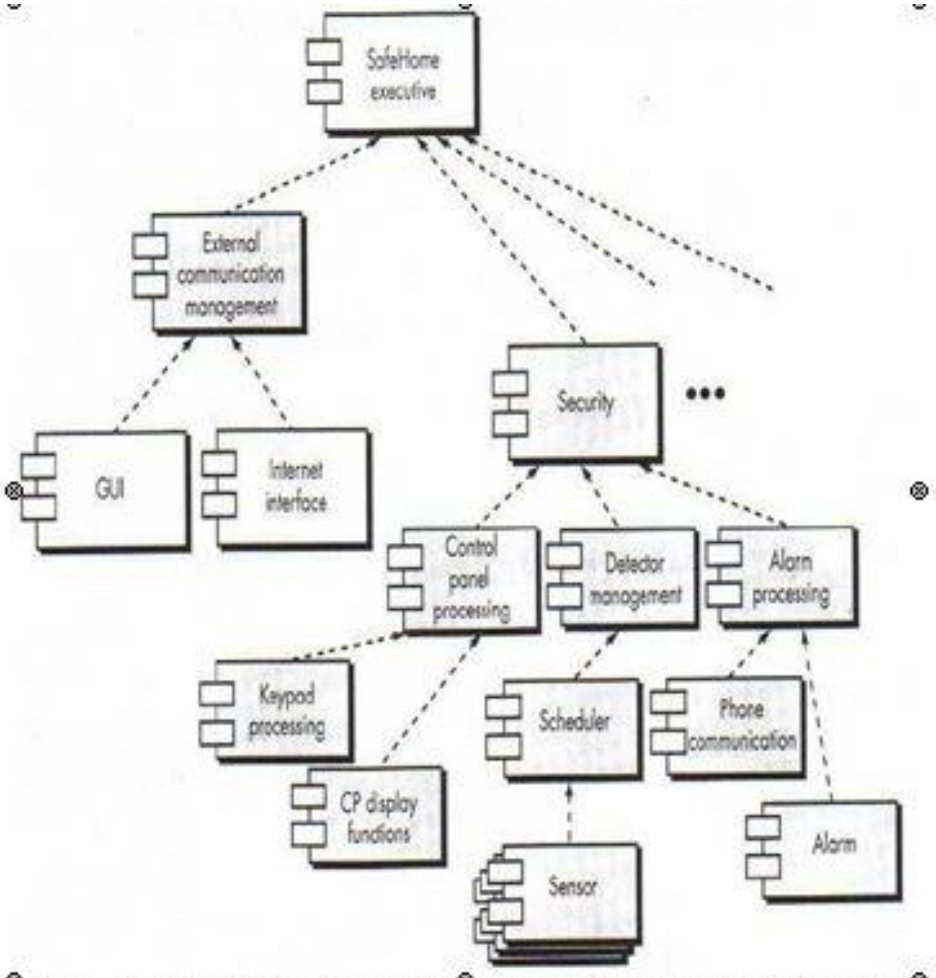


Figure: An Instantiation of security function with component elaboration

➢ An Architecture Trade-Off Analysis Method:

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

1. ***Collect scenarios.*** A set of use cases is developed to represent the system from the user's point of view.
2. ***Elicit requirements, constraints, and environment description.*** This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. ***Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.*** The architectural style(s) should be described using one of the following architectural views:
   - ***Module view*** for analysis of work assignments with components

andthe degree to which information hiding has been achieved.
- ***Process view*** for analysis of system performance.
- ***Data flow view*** for analysis of the degree to which the architecture meets functional requirements.

4. ***Evaluate quality attributes by considering each attribute in isolation.*** The number of quality attributes chosen for analysis is a function of the timeavailable for review and the degree to which quality attributes are relevant to the sys-tem at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. ***Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.*** This can be accomplished bymaking small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*
6. ***Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.*** The SEI describes this approachin the following manner.

   Once the architectural sensitivity points have been determined, findingtrade-off points is simply the identification of architectural elements to

   which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The number of servers, then, is a trade-off point with respect to this architecture.

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

- Architectural Complexity:

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

system. Zhao suggests three types of dependencies:

Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.

Flow Dependencies represent dependence relationships between producers and consumers of resource. For example for two components u and v, if u must complete before controls flows into v or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.

Constrained dependencies represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, if u andv cannot execute at the same time then there exists a constrained dependence relationship between u and v.

The sharing and flow dependencies noted by Zhao are similar to the concept of coupling. Coupling is an important design concept that is applicable at the architectural level and at the component level.

- Architectural Description Languages:

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

Once descriptive, language-based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.