UNIT - III

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

THE DESIGN PROCESS AND DESIGN QUALITY:

Design Engineering encompasses the **set of principles, concepts, and practices** that lead to the development of a high-quality system or product. The importance of software design can be stated with a single word—*quality*.

Design Engineering is an iterative process through which requirements are translated into a **"blueprint" for constructing the software**. 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.

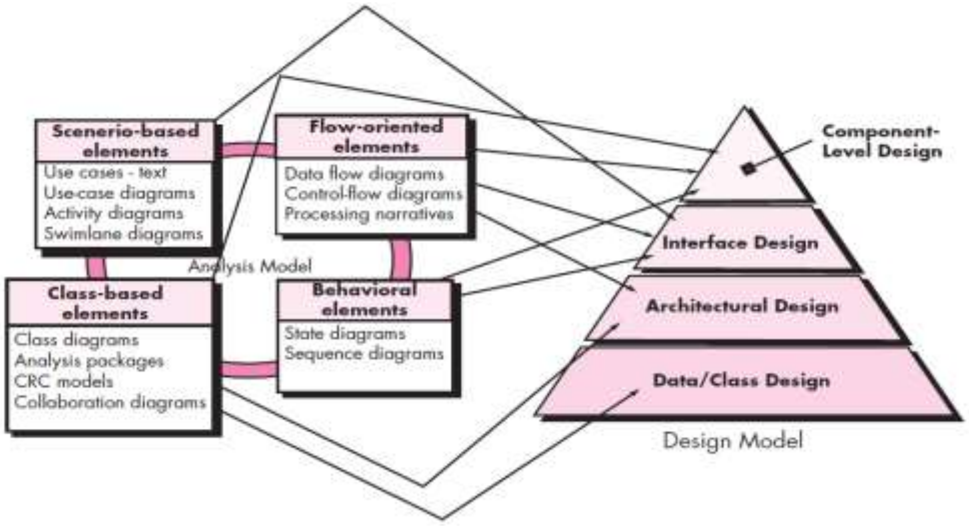


Fig.: Translating the requirements model into the design model

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).

The design model provides detail about **software architecture, data structures, interfaces, and components** that are necessary to implement the system. The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost.

Three characteristics of a good design/ Goal of Design process:

The **goal of design** is to produce a model or representation that exhibits firmness, commodity, and delight.

Firmness: A program should not have any bugs that inhibit its function.

Commodity: A program should be suitable for the purposes for which it was intended.

Delight: The experience of using the program should be a pleasurable one.

1. 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**.
2. The design must be a readable, understandable guide for those who generate **code** and for those who **test** and subsequently **support** the software.
3. The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an **implementation perspective**.

Quality Guidelines:

In order to evaluate the quality of a design representation (or) **to achieve goals**, the software team must establish technical criteria for good design. Consider the following 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 characteristics**

(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.

Quality Attributes:

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.

DESIGN CONCEPTS :

Fundamental Software Design Concepts provide the software designer with a foundation from which more sophisticated design methods can be applied.

1. Abstraction - Procedural abstraction , Data abstraction

2. Architecture - Structural properties, Extra functional properties, Families of similar systems - Structural models, Framework models, Dynamic models, Process models, Functional models,

3. Patterns - Design structure

4. Modularity - Effort (cost) decreases as the total number of modules increases

5. Information Hiding - Defining a set of independent modules

6. Functional Independence - Cohesion and Coupling

7. Refinement - Step wise refinement, Low level abstraction

8. Aspects - Crosscutting concern

9. Refactoring - Reorganization technique

10. Design Classes - Refine the analysis classes

1. Abstraction :

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 and 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. Ex. ***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, weight, dimensions, opening mechanism). It follows that the **procedural abstraction** *open* would make use of information contained in the attributes of the **data abstraction** door**.**

2. Architecture :

Software 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.

Properties 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. ex. ***Structural models***

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, reliability, security, adaptability, and other system characteristics. ex. ***Dynamic models*** address the behavioral aspects of the program architecture. **F*unctional models*** can be used to represent the functional hierarchy of a system. ***Process models*** focus on the design of the business or technical process that the system must accommodate.

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**. ex. ***Framework models*** increase the level of design abstraction by attempting to **identify repeatable architectural design** frameworks that are encountered in similar types of applications.

3. Patterns :

Design pattern describes a design structure that solves a particular design problem within a specific context.

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

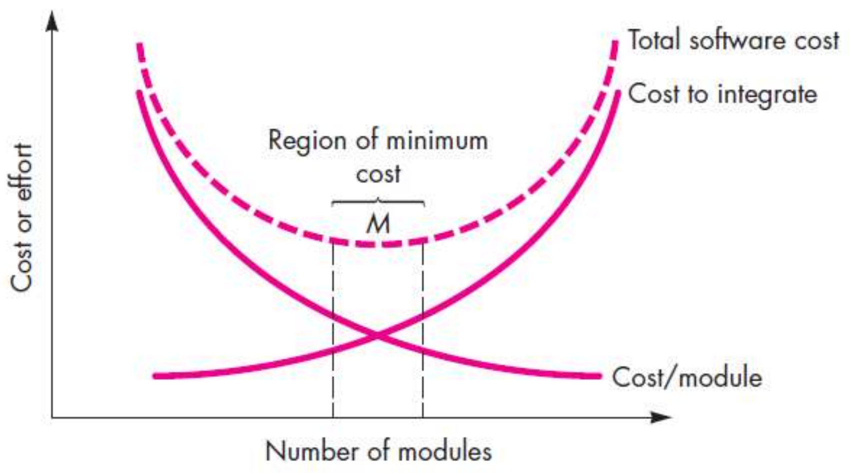
(1) whether the pattern is applicable to the current work,

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

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

4. Modularity :

Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.



Referring to Figure, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. Under modularity or over modularity should be avoided. You modularize a design so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

5. Information Hiding :

The principle of information hiding suggests that 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. ie., design decisions that each hides from all others

Hiding implies that effective modularity can be achieved by defining a set of **independent** modules. The use of information hiding for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.

6. Functional Independence:

The concept of functional independence is a separation of concerns, modularity, and the concepts of abstraction and information hiding. 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.

<u>7. Refinement :</u>

Refinement is actually a process of *elaboration.* You begin with a statement of function (or description of information) 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 information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs. Stepwise refinement is a top-down design strategy.

<u>8. Aspects :</u>

An *aspect* is a representation of a crosscutting concern. Consider two requirements, *A* and *B.* Requirement *A crosscuts* requirement *B. ie., B* cannot be satisfied without taking *A* into account. For example, the design representation, *B* of the requirement received concern that, *a registered user must be validated prior to using SafeHome* WebApp.

It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. The design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

<u>9. Refactoring :</u>

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

<u>10. Design Classes :</u>

The requirements model defines a set of analysis classes. The level of abstraction of an analysis class is relatively high. As the design model evolves, you will define a set of *design classes* that refine the analysis classes that present significantly **more technical detail** as a guide for implementation.

<u>Five different types of design classes:</u>

• ***User interface classes*** define all abstractions that are necessary for human computer interaction (HCI).

• ***Business domain 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.

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. 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 (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time.

THE DESIGN MODEL :

The design model can be viewed in two different dimensions. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

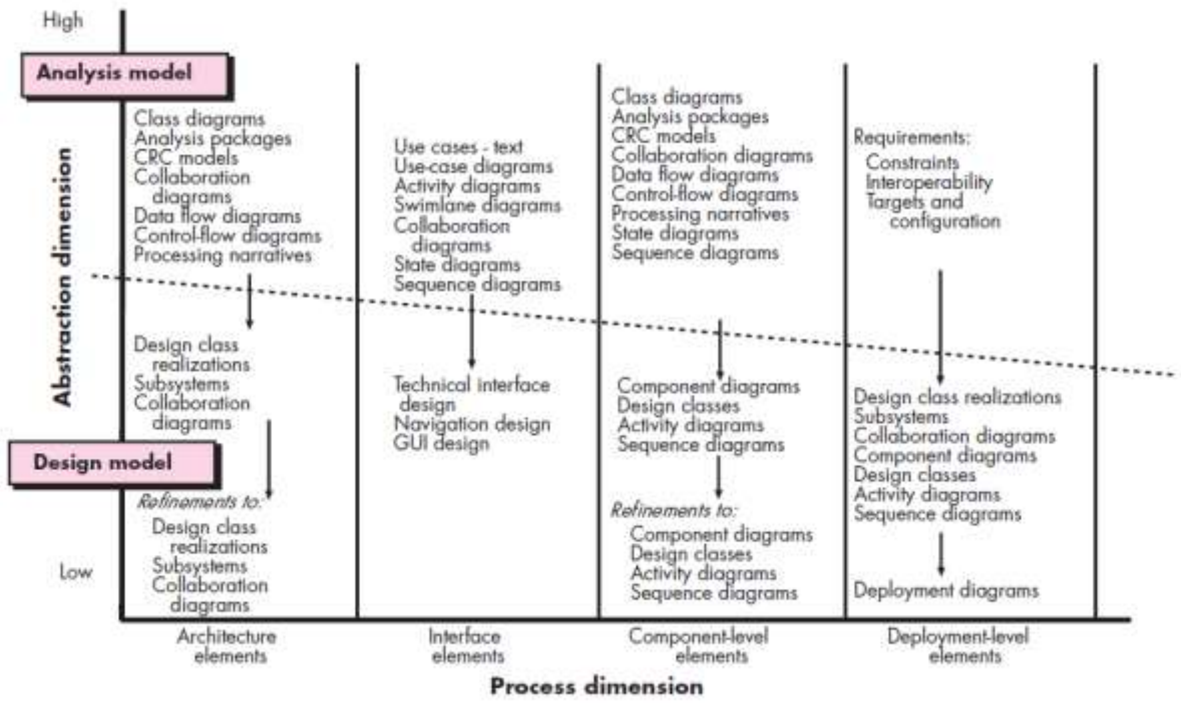


Fig. : Dimensions of the Design Model

Referring to Figure, the dashed line indicates the boundary between the analysis and design models. 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 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.

The model elements indicated 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 is usually delayed until the design has been fully developed.

1. Data Design Elements:

Data design creates a model of data that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations.

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 (derived as part of requirements engineering) 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.

2. Architectural Design Elements:

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

The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model. Each subsystem may have its own architecture.

3. Interface Design Elements:

The interface design for software is analogous to a set of **detailed drawings** (and specifications) for the doors, windows, and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan.

They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor's presence, and how a security system is to be installed. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components.

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

4. Component-Level Design Elements:

The component-level design for software is the equivalent to a set of detailed drawings and specifications for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room. 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).

5. Deployment-Level Design Elements:

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

The personal computer houses subsystems that implement security, surveillance, home management, and communications features. Each subsystem would be elaborated to indicate the components that it implements.

PATTERN BASED SOFTWARE DESIGN :

A software Engineer should look for every opportunities in the design process to reuse existing design patterns (when they meet the needs of the design) rather than creating new ones.

Describing a Design Pattern

A description of the design pattern may consider a set of design forces.

- Design forces describe **nonfunctional requirements** (eg. ease of maintainability, portability) associated the software for which the pattern is to be applied.
- Design forces define the **constraints** that may restrict the manner in which the design is to be implemented.
- Design forces describe the **environment and conditions** that must exist to make the design pattern applicable.
- The pattern characteristics (classes, responsibilities, and collaborations) indicate the **attributes of the design** that may be adjusted to enable the pattern to accommodate a variety of problems. These attributes represent characteristics of the design that can be searched(eg. via a database) so that an appropriate pattern can be found.
- Finally, guidance associated with the use of a design pattern provides an indication of the **ramifications** of design decisions.

- The names of design patterns must be **meaningful**.

<u>Using Patterns in Design:</u>

Architectural Patterns - These patterns define the overall structure of the software, indicate the relationships among sub systems and software components, and define the rule for specifying relationships among the elements(classes, packages, components, subsystems) of the architecture.

Design Patterns - These patterns address a specific element of the design such as an aggregation of components to solve some design problem, relationships among components, or the mechanisms for effecting component-to-component communication.

Idioms - Sometimes called coding patterns, these language specific patterns generally implement an algorithmic element of a component, a specific interface protocol, or a mechanism for communication among components.

<u>Frameworks :</u>

Framework provide an implementation specific skeletal infrastructure for design work. The designer may select a "reusable mini-architecture" that provides the generic structure and behaviour for a family of software abstractions.

A framework is not an architectural pattern, but rather a skeleton with a collection of "plug points" (also called slots) that enable it to be adapted to a specific problem domain. The plug points enable a designer to integrate problem specific classes or functionality within the skeleton. In an object oriented context, a framework is a collection of cooperating classes.

CREATING AN ARCHITECTURAL DESIGN :

Architectural design represents the structure of data and program components that are required to build a computer-based system. Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels.

In addition, component properties and relationships (interactions) are described.

- The “system architect” selects an appropriate architectural style from the requirements derived during software requirements analysis.
- A database or data warehouse designer creates the data architecture for a system.
- The software engineer can design both data and architecture.

<u>SOFTWARE ARCHITECTURE :</u>

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

Importance of software architecture:

The representations of software architecture enables to

(1) Analyze the effectiveness of the design in meeting its stated requirements

(2) Consider architectural alternatives at a stage when making design changes

(3) Reduce the risks associated with the construction of the software.

(4) Communicate between stakeholders

(5) highlights early design decisions

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. I can design 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. Therefore, you cannot mix “architecture” and “design” with each other. So Design begins with a consideration of architecture.

DATA DESIGN :

The data design action translates data objects defined as part of the analysis model into

1. Database architecture at the application level.
2. Data structures at the Component level

Data design at Architectural level / Database architecture at the application level:

A **data warehouse** is a separate data environment that encompasses all data used by a business. 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. The **data mining** techniques, also called knowledge discovery in databases (KDD) navigate through databases to extract appropriate business level information.

Data design at component level / Data structures 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. The following principles form a basis for a component-level data design approach that can be integrated into both the analysis and design activities.

Set of principles followed for data specification:

1. The systematic analysis principles applied to function and behavior should also be applied to data. We spend much time and effort **deriving**, **reviewing**, and specifying functional requirements and preliminary design. Representations of data 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. An alternative data organization may lead to better results.

2. The data structure is to be manipulated in a number of major software functions. Upon evaluation of the operations performed on the data structure, an **abstract data type** is defined for use in subsequent software design.

3. A **data dictionary** should be established and used to define both data and program design. A data dictionary explicitly represents the relationships among data objects and the constraints on the elements of a data structure.

4. 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 by top-down approach during component level design.

5. The representation of data structure should 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. This principle alludes to the importance of these concepts as well as "the importance of separating the logical view of a data object from its physical view".

6. A **library** of useful data structures and the operations that may be applied to them should be developed. Data structures can be designed for reusability. A library of data structure templates can reduce both specification and design effort for data.

7. A **software design and programming language** should support the specification and realization of abstract data types.

ARCHITECTURAL STYLES AND PATTERNS:

The architectural style is also a **template for construction**. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the style—a “center hall colonial”—guides the builder in his work. The intent of architectural style is to establish a structure for all components of the system.

Each architectural style describes a system category that encompasses

(1) a set of **components** (e.g., a database, computational modules) that perform a function required by a system;

(2) a set of **connectors** that enable “communication, coordination and cooperation” among components;

(3) **constraints** that define how components can be integrated to form the system; and

(4) **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An architectural Patterns can be used in conjunction with an architectural style to shape the overall structure of a system. An architectural pattern differs from an architectural style in a number of fundamental ways:

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

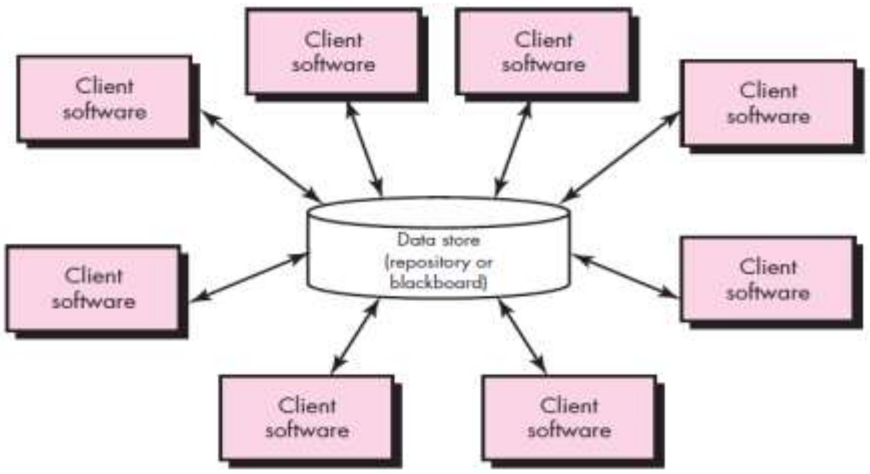
(2) a pattern **imposes a rule on the architecture**, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency);

(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).

A Brief Taxonomy of Architectural Styles

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Client software accesses a central repository.

Data-centered architecture

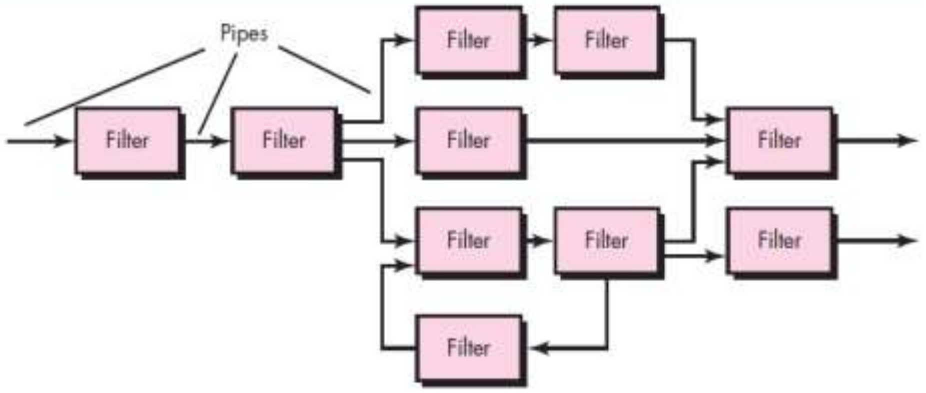


In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a “blackboard” that sends notifications to client software when data of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

Data-centered architectures promote *integrability*. 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). Client components independently execute processes. 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).

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 has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. The filter does not require knowledge of the workings of its neighboring filters.

FIGURE 9.2
Data-flow architecture

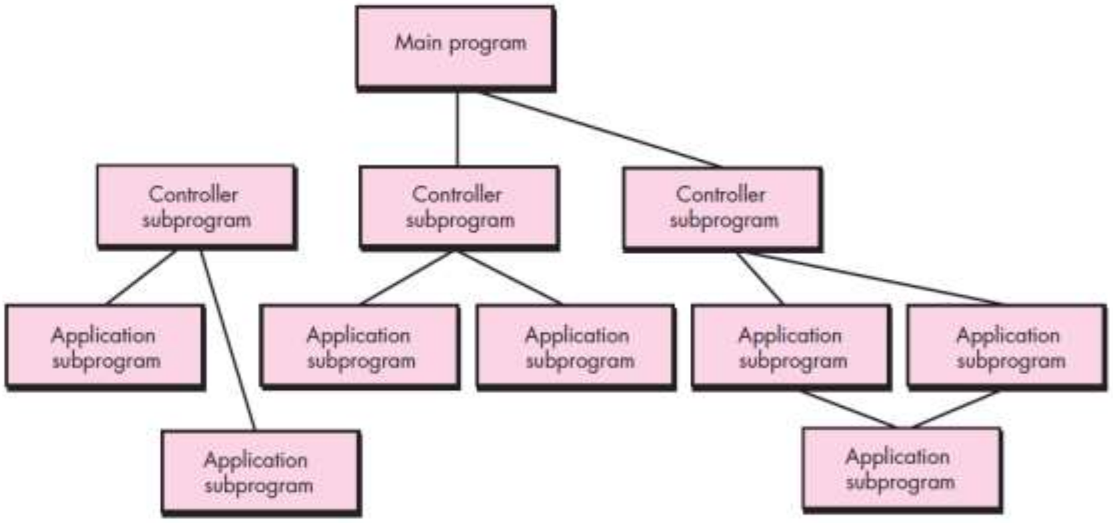


Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A two substyles exist within this category:

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

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

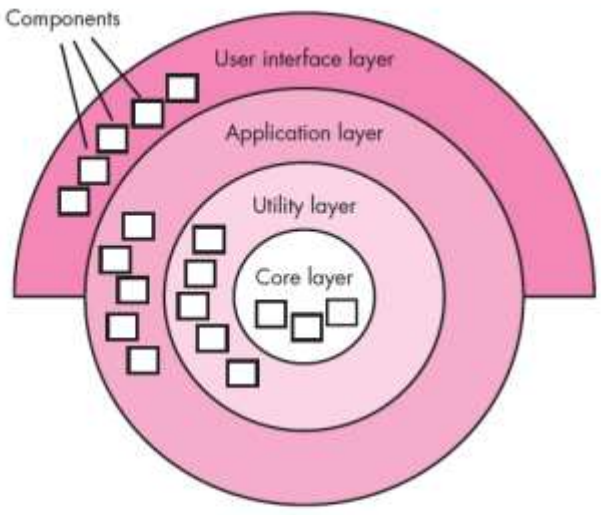
FIGURE 9.3 Main program/subprogram architecture



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

Layered architectures. The basic structure of a layered architecture is illustrated in Figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components

FIGURE 9.4 Layered architecture



service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

ARCHITECTURAL PATTERNS

If a house builder decides to construct a center-hall colonial, there is a single architectural style that can be applied. The details of the style (e.g., number of fireplaces, façade of the house, placement of doors and windows) can vary considerably, but once the decision on the overall architecture of the house is made, the style is imposed.

For example,

Kitchen pattern : It **collaborates with address problems** associated with the storage and preparation of food, the tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room. In addition, the pattern might address problems associated with countertops, lighting, wall switches, a central island, flooring, and so on. Every design can be conceived within the context of the "solution" suggested by the Kitchen pattern.

Architectural pattern domains:

- **Access control.** There are many situations in which access to data, features, and functionality delivered by an application is limited to specifically defined end users.
- **Concurrency.** Many applications must handle multiple tasks in a manner that simulates **parallelism** (i.e., this occurs whenever multiple "parallel" tasks or components are managed by a single processor). For example, one approach is to use an Operating System- **Process Management pattern** that provides built-in OS features that allow components to execute concurrently. The pattern also incorporates OS functionality that manages communication between processes, scheduling, and other capabilities required to achieve concurrency. Another approach might be to define a **task scheduler** at the application level.
- **Distribution.** The distribution problem addresses the manner in which systems or components within systems communicate with one another in a distributed environment. The most common architectural pattern established to address the distribution problem is the **Broker pattern**. A broker acts as a "middle man" between the client component and a server component. The client sends a message to the broker (containing all appropriate information for the communication to be effected) and the broker completes the connection.

- **Persistence.** Persistent data are stored in a database or file and may be read or modified by other processes at a later time. In general, two architectural patterns are used to achieve persistence—a **Database Management System pattern** that applies the storage and retrieval capability of a DBMS to the application architecture. An **Application Level- Persistence pattern** that builds persistence features into the application architecture (e.g., word processing software that manages its own document structure).

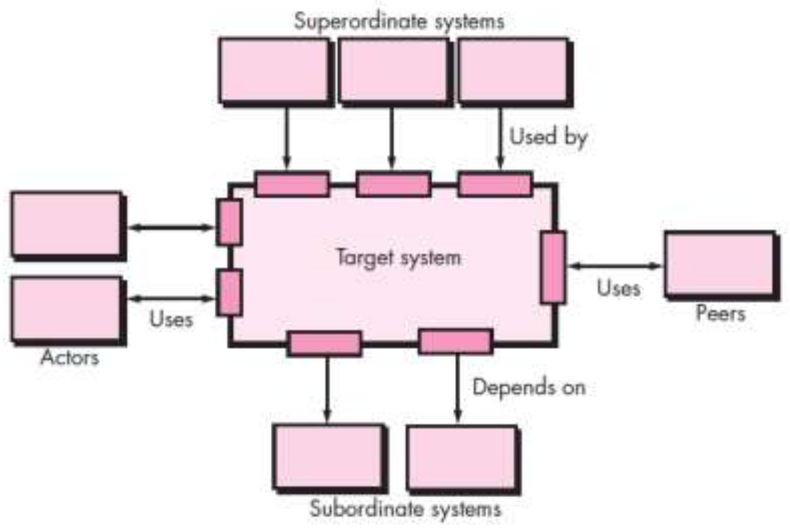
ARCHITECTURAL DESIGN :

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

FIGURE 9.5
Architectural context diagram
Source: Adapted from [Bos00].

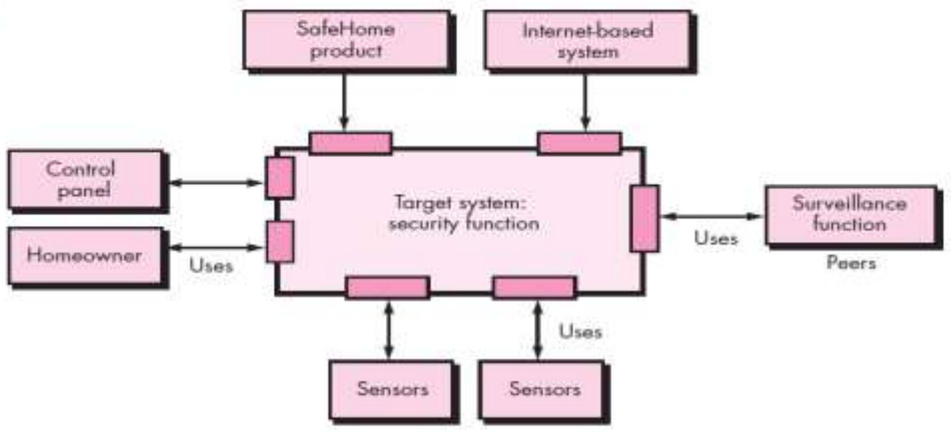


Referring to the figure, systems that interoperate with the ***target system*** (the system for which an architectural design is to be developed) are represented as

- *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- *Peer-level systems*—those systems that **interact** on a peer-to-peer basis i.e., information is either produced or consumed by the peers and the target system.
- *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.
- Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

FIGURE 9.6

Architectural context diagram for the *SafeHome* security function



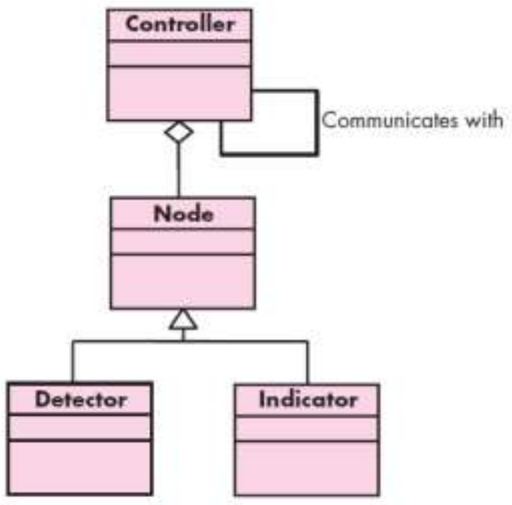
Defining Archetypes

An *archetype* is a class or pattern that represents a **core abstraction** that is critical to the design of an architecture **for the target system**. The target system architecture is composed of these archetypes, which represent stable elements of the architecture. Each of archetypes is

FIGURE 9.7

UML relationships for *SafeHome* security function archetypes

Source: Adapted from [Bos00].



depicted using UML notation as shown in Figure.

The *SafeHome* home security function might define the following archetypes:

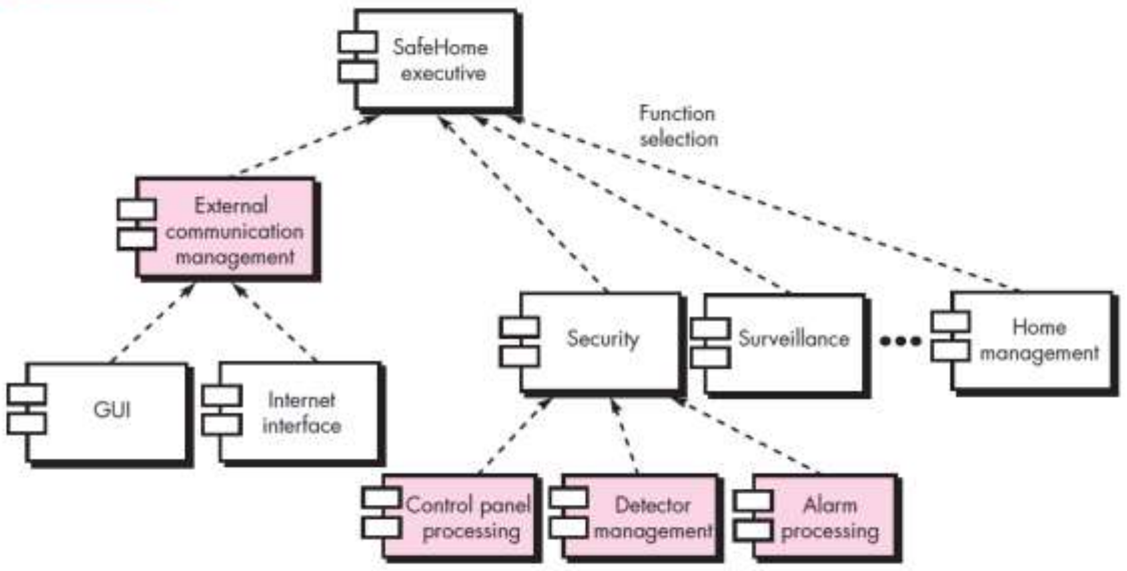
- **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors(Detector) and (2) a variety of alarm, flashing lights, bell (indicators).
- **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

<u>Refining the Architecture into Components:</u>

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

The **application(business) domain** is one source for the derivation and refinement of components. Another source is the **infrastructure domain**. The interfaces depicted in the architecture context diagram imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed. The overall architectural structure (represented as a UML component diagram) is illustrated in Figure.

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



The set of top-level components that address the following functionality:

- *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- *Control panel processing*—manages all control panel functionality.
- *Detector management*—coordinates access to all detectors attached to the system.
- *Alarm processing*—verifies and acts on all alarm conditions.

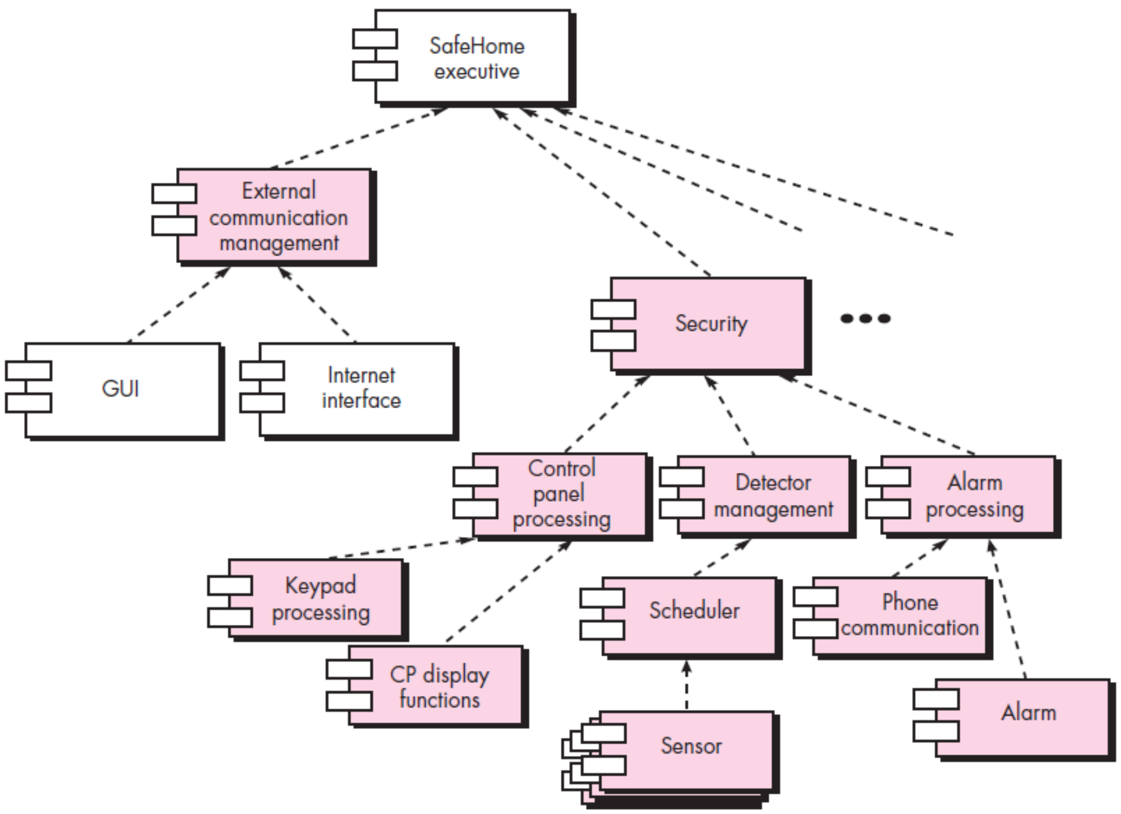
Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture.

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.

To accomplish this, an actual instantiation of the architecture is developed. Components are elaborated to show additional detail.

FIGURE 9.9 An instantiation of the security function with component elaboration


SafeHome executive
External communication management
GUI
Internet interface
Security
•••
Control panel processing
Detector management
Alarm processing
Keypad processing
CP display functions
Scheduler
Phone communication
Sensor
Alarm