UNIT – 4

Unit III:

Design Concepts: Design with Context of Software Engineering, The Design Process, Design Concepts, The Design Model.

Architectural Design: Software Architecture, Architecture Genres, Architecture Styles, Architectural Design, Assessing Alternative Architectural Designs, Architectural Mapping Using Data Flow.

Component-Level Design: Component, Designing Class-Based Components, Conducting Component-level Design, Component Level Design for WebApps, Designing Traditional Components, Component-Based Development.

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model, the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Who does it? Software engineers conduct each of the design tasks. Why is it important? Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers. Design is the place where software quality is established.

What are the steps? Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed. Each of these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.

What is the work product? A design model that encompasses architectural, interface, component level, and deployment representations is the primary work product that is produced during software design.

How do I ensure that I've done it right? The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established.

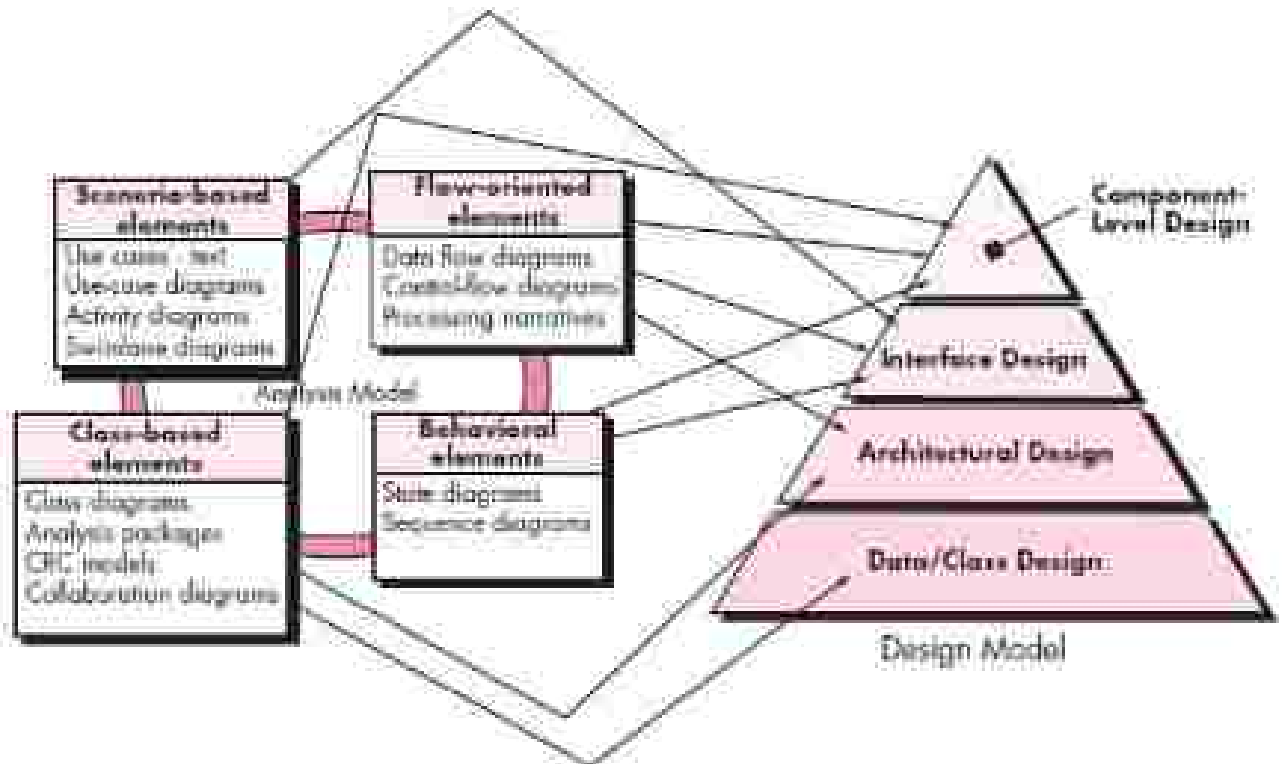


Fig 8.1: Translating the requirements model into the design model

communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

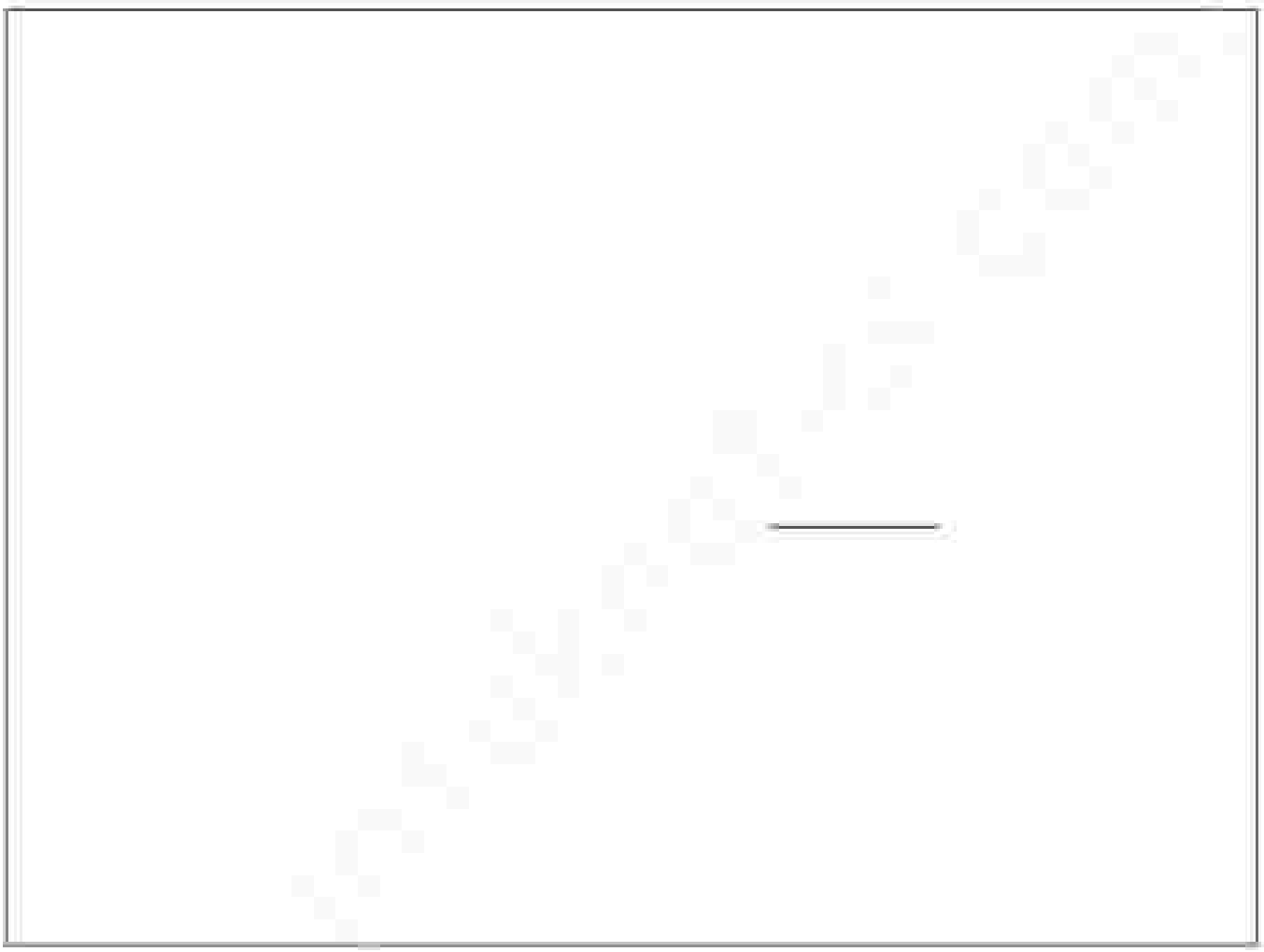
Functional Independence: The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with function and an to excessive interaction with other modules. Stated another way, *you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure*. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: **cohesion** and ***coupling.***
Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

A **cohesive module** performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software , caused when errors occur at one location and propagate throughout a system.

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



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

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

Why Is Architecture Important?: Three key reasons that software architecture is important:
• Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
• The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
• Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

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

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

The IEEE standard defines an architectural description (AD) as "*a collection of products to document an architecture.*" The description itself is represented using multiple views, where each view is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

Architectural Decisions:

To develop each view the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern

ARCHITECTURAL GENRES

the architectural genre will often dictate the specific architectural approach to the structure that must be built

Grady Booch

Artificial intelligence

Commercial and nonprofit

Communications

Content authoring

Devices

Entertainment and sports

Financial

Games

Government

Industrial

Legal

Medical

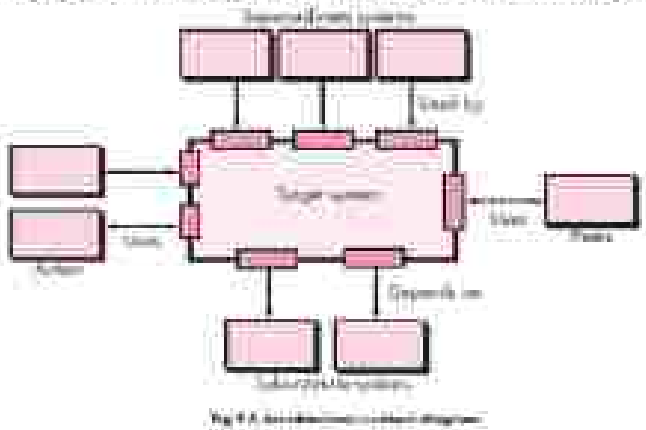
Military

Operating systems

Platforms

Peer-level systems—those systems that interact on a peer-to-peer basis (i.e. information is either produced or consumed by the peers and the target system.

* *Actors—entities* (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Each of these external entities communicates with the target system through an interface (the small shaded rectangles).



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

The following are the archetypes for safeHome: Node, Detector, Indicator, Controller.

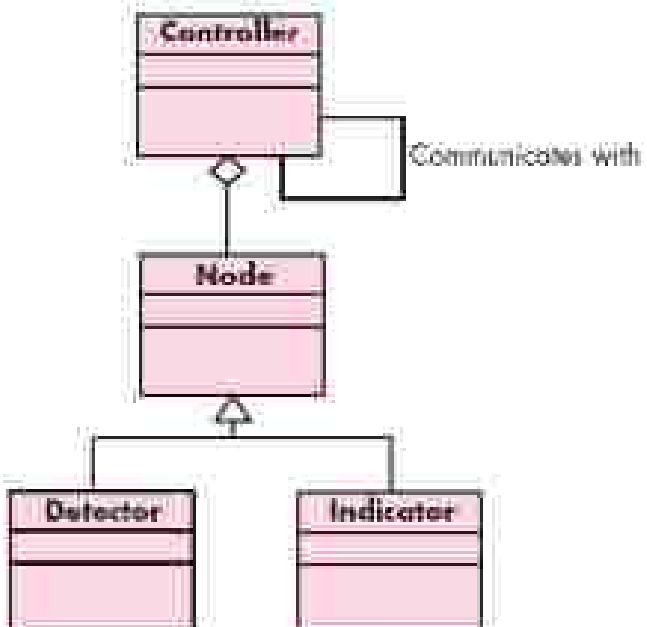
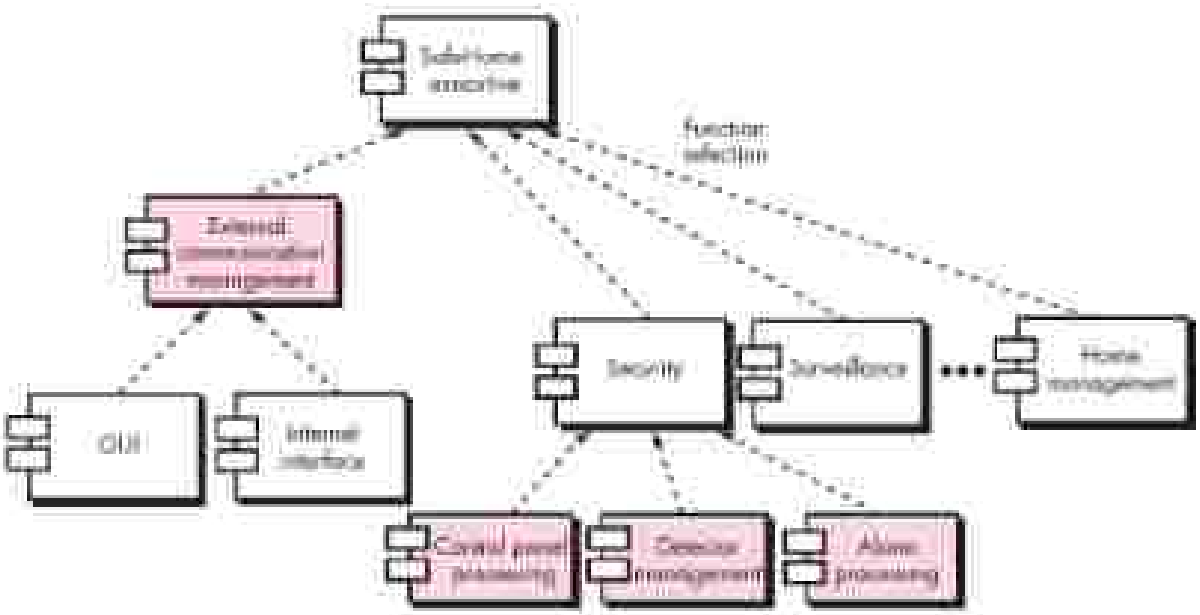


Fig 7.7: UML relationships for safehome function Archetypes


Function selection
Security
Surveillance
GUI
Internal interface

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

Standard data. The application domain should be investigated and standard global data structures (e.g., file structures or a complete database) should be identified. All design components can then be characterized to make use of these standard data structures.

Standard interface protocols. Three levels of interface protocol should be established: the nature of intramodular interfaces, the design of external technical (nonhuman) interfaces, and the human-computer interface.

Program templates. An architectural style is chosen and can serve as a template for the architectural design of a new software. Once standard data, interfaces, and program templates have been established, you have a framework in which to create the design. New components that conform to this framework have a higher probability for subsequent reuse.

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

A reusable software component can be described in many ways, but an ideal description encompasses the 3C model *concept, content and context*. The *concept* of a software component is "a *description of what the component does*". The interface to the component is fully described and the semantics represented within the context of pre- and post conditions is identified. The *content* of a component describes how the concept is realized. The *context* places a reusable software component within its domain of applicability.

A reuse environment exhibits the following characteristics:

A component database capable of storing software components and the classification information necessary to retrieve them.

A library management system that provides access to the database.

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

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

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

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

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