UNIT III

SOFTWARE DESIGN

Design process – Design Concepts-Design Model– Design Heuristic – Architectural Design – Architectural styles, Architectural Design, Architectural Mapping using Data Flow- User Interface Design: Interface analysis, Interface Design –Component level Design: Designing Class based components, traditional Components.

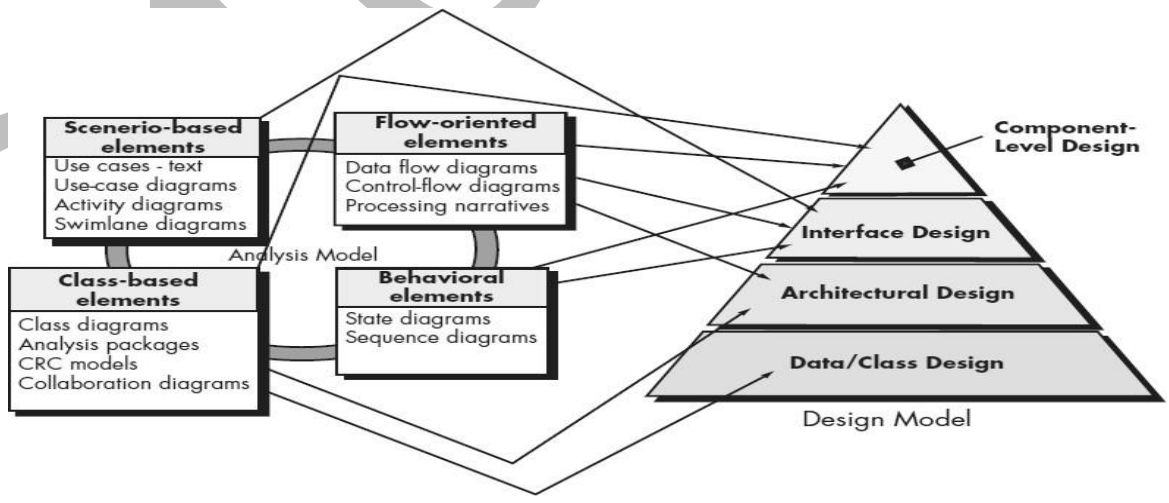
3.1 DESIGN PROCESS

3.1.1 Software Design and Software Engineering

- The flow of information during software design is illustrated in Figure 3.1.
- The **data design**transforms the information domain model created during analysis into the data structures that will be required to implement the software.
- The **architectural design**defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system,
- The architectural design representation— the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.
- The **interface design**describes how the software communicates within itself, 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 PSPEC, CSPEC, and STD serve as the basis for component design.
- Design is the only way that we can accurately translate a customer's requirements into a finished



software product or system.

Figure 3.1 Translating the analysis model into a software design

3.1.2 The Design Process

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

3.1.2.1 Design and Software Quality

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

1. A design should exhibit an architectural structure that

(1) has been created using recognizable design patterns,

(2) is composed of components that exhibit good design characteristics

(3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and sub functions.

3. A design should contain distinct representations of data, architecture, interfaces, and components (modules).

4. A design should lead to data structures that are appropriate for the objects 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 modules 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.

Quality Attributes:

- 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

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.

3.1.2.2 The Evolution of Software Design

The evolution of software design is a continuing process that has spanned the past four decades.

- Modular programs and methods for refining software structures in a top-down manner.
- Procedural aspects of design definition called structured programming.
- The translation of data flow or data structure into a design definition.
- Object-oriented approach to design derivation.

Like this many design methods, growing. Each software design method introduces unique heuristics and notation, Yet, all of these methods have a number of common characteristics:

(1) A mechanism for the translation of analysis model into a design representation,

(2) A notation for representing functional components and their interfaces,

(3) Heuristics for refinement and partitioning,

(4) Guidelines for quality assessment.

Regardless of the design method that is used, a software engineer should apply a set of fundamental principles and basic concepts to data, architectural, interface, and Component- level design.

3.2 DESIGN CONCEPTS

Design concepts provides the software designer with a foundation from which more sophisticated design methods can be applied and helps the software engineer to answer the following questions:

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

Fundamental software design concepts provide the necessary framework for "getting it right."

- Abstraction
- Architecture
- Patterns
- Separation of Concerns
- Modularity
- Information Hiding
- Functional Independence
- Refinement
- Aspects
- Refactoring
- Object-Oriented Design Concepts
- Design Classes

3.2.1 Abstraction

- Each step in the software process is a refinement in the level of abstraction of the software solution.
- **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
- **An example of data abstraction** called **door**. Like any data object, the data abstraction for **door** would encompass set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).
- It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction **door**.
- **Control abstraction** is the third form of abstraction used in software design. Control abstraction implies a program control mechanism without specifying internal details.
  - An example of a control abstraction is the synchronization semaphore] used to
  - coordinate activities in an operating system.

3.2.2 Software Architecture

Software architecture means "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". A set of architectural patterns enable a software engineer to reuse design level concepts.

Properties that should be specified as part of an architectural design:

a) Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

b) Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

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

c) 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.

Model	Functioning
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 behavioural 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.

3.2.3 Patterns

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

3.2.4 Separation of Concerns

- Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A concern is a feature or behaviour that is specified as part of the requirements model for the software. By separating concerns into smaller and therefore more manageable pieces, a problem takes less effort and time to solve.

3.2.5 Modularity

- Software is divided into separately named and addressable components, often 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 (i.e., a large program composed of a single module) 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.

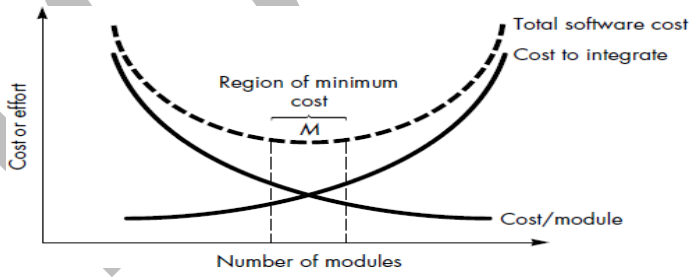


Figure 3.2 Modularity and software cost

- The curves shown in Figure 3.2(Modularity and software cost) do provide useful guidance when modularity is considered.
- Another important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system.
- Five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

a) **Modular decomposability.** If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

b) **Modular composability.** If a design method enables existing (reusable)design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

c) **Modular understandability.** If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.

d) **Modular continuity.** If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.

e) **Modular protection**. If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

3.2.6 Information Hiding

- Modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.
- Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.
- **The benefit of information hiding:** 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 the software.

3.2.7 Functional Independence

- The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.
- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- 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.
- Independence is assessed using two qualitative criteria: **cohesion and coupling.**
  Coupling is an indication of the relative interdependence among modules. **Cohesion** is a natural extension of the information-hiding concept

3.2.8 Refinement

- Refinement is actually a process of elaboration.
- We begin with a statement of function (or description of information) that is defined at a high level of abstraction. I.e. 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.
- Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

3.2.9 Aspects

- An aspect is a representation of a crosscutting concern.
- It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur.
- An aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components.

- To accomplish this, 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.

3.2.10 Refactoring

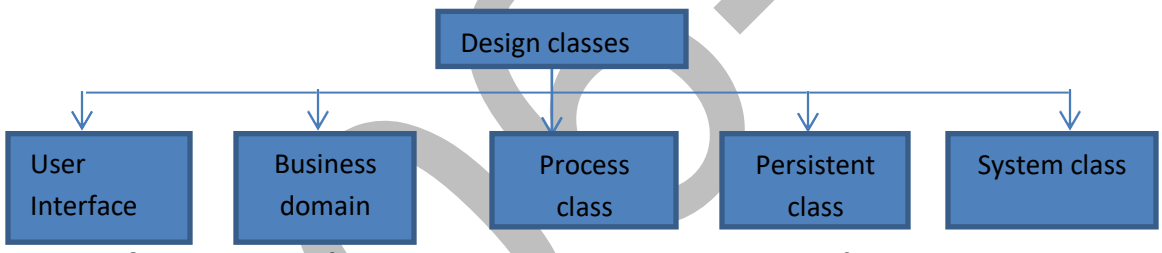
- Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behaviour.
- Fowler defines "Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code [design] improves its internal structure." **Benefits of refactoring:**
- The **redundancy** can be achieved.
- **Inefficient algorithms** can be eliminated or can be replaced by efficient one.
- Poorly constructed or **inaccurate data structures** can be removed or replaced.

3.2.11 Object-Oriented Design Concepts:

- The object-oriented (OO) paradigm is widely used in modern software engineering.
- OO design concepts such as classes and objects, inheritance, messages, and polymorphism.

3.2.12 Design Classes

- A set of design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- There are five different types of design classes.



- ✓ **User interface classes** define all abstractions that are necessary for human computer interaction (HCI).
- ✓ **Business domain classes** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- ✓ **Process classes** implement lower-level business abstractions required to fully manage the business domain classes.
- ✓ **Persistent classes** represent data stores (e.g., a database) that will persist beyond the execution of the software.
- ✓ **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world

3.3 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

Referring to Figure 3.3, 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.

Dimensions of the design model

3.3.1 Data Design Elements

- **Data design** referred to as data architecting creates a model of data and/or information that is represented at a **high level of abstraction** (the customer/user's view of data).
- This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

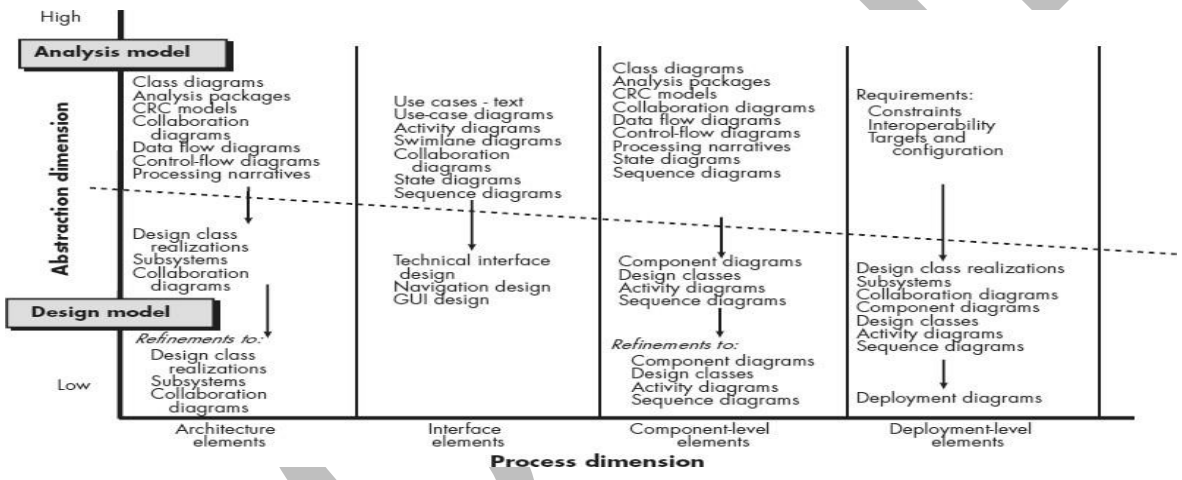


Figure 3.3 Boundary between Analysis and Design Models

- The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At the **application level,** the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- At the **business level,** the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

3.3.2 Architectural Design Elements

- Architectural design elements give us an overall view of the software. Architectural model can be built using following sources-
- Data flow models or class diagrams
- Information obtained from application domain
- Architectural patterns and styles.

3.3.3 Interface Design Elements

The interface design for software is analogous to a set of detailed drawings.

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

3.3.4 Component -Level Design Elements

- The component-level design for software is the equivalent to a set of detailed drawings (and specifications).
- The component-level design for software fully describes the internal detail of each software component.
- Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 3.4.
- In this figure, a component named Sensor Management (part of the Safe Home security function) is represented.
- A dashed arrow connects the component to a class named Sensor that is assigned to it.
- The Sensor Management component performs all functions associated with Safe Home sensors including monitoring and configuring them.

Figure 3.4 A UML component diagram

3.3.5 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.
- For example, the elements of the SafeHome product are configured to operate within three primary computing environments—a home-based PC, the SafeHome control panel, and a server housed at CPI Corp. (providing Internet-based access to the system).

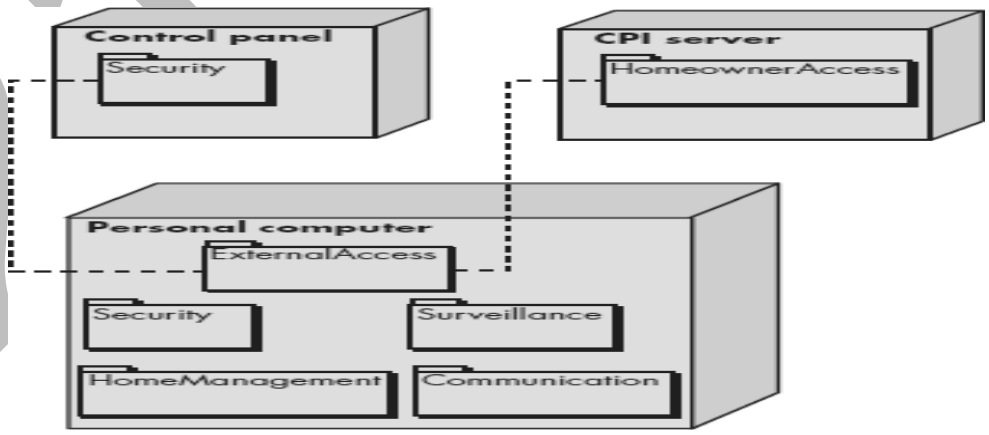


Figure 3.5 A UML deployment diagram

3.4 DESIGN HEURISTICS

The program structure can be manipulated according to the following set of heuristics:

1. **Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.**

i) Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence.
ii) An exploded module becomes two or more modules in the final program structure.
iii) An imploded module is the result of combining the processing implied by two or more modules.
iv) An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module.
v) When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

2. **Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.**

i) The structure shown inside the cloud in below Figure 3.6 does not make effective use of factoring. All modules are “pancaked” below a single control module.
ii) In general, a more reasonable distribution of control is shown in figure 3.7 the upper structure.
iii) The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

3. **Keep the scope of effect of a module within the scope of control of that module.**

i) The scope of effect of module e is defined as all other modules that are affected by a decision made in module e.
ii) The scope of control of module e is all modules that are subordinate and ultimately subordinate to module e.
iii) Referring to the below Figure 3.6, if module e makes a decision that affects module r,we have a violation of this heuristic, because module r lies outside the scope of control of module e.

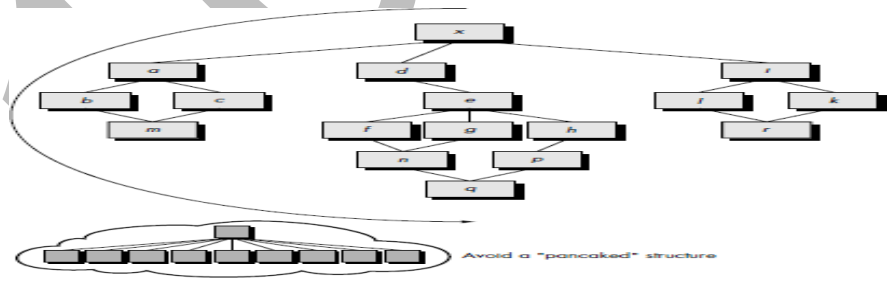


Figure 3.6 Program structures

4. **Evaluate module interfaces to reduce complexity and redundancy and improve**

i) **Consistency**. Module interface complexity is a prime cause of software errors.
ii) Interfaces should be designed to pass information simply and should be consistent with the function of a module.

iii) Interface inconsistency is an indication of low cohesion. The module in question should be re-evaluated.

5. Define modules whose function is predictable, but avoid modules that are overly restrictive.

i) A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details.

ii) Modules that have internal "memory" can be unpredictable.

iii) A module that restricts processing to a single sub function exhibits high Cohesion.

6. Strive for "controlled entry" modules by avoiding "pathological connections."

i) This design heuristic warns against content coupling.

ii) Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled.

iii) Pathological connection refers to branches or references into the middle of a module.

3.5 ARCHITECTURAL DESIGN

3.5.1 SOFTWARE ARCHITECTURE

Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

3.5.1.1 What Is 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 a software engineer to

  (1) Analyse 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

  (3) Reducing the risks associated
- The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components.
- 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.
- Software architecture considers two levels of the design pyramid

• Data design and architectural design.

- Data design enables us to represent the data component of the architecture.
- Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

3.5.1.2 Why is Architecture Important?

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

• The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

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

3.6 ARCHITECTURAL STYLES

- The architectural model or style is a pattern for creating the system architecture for given problem.
- The software that is built for computer-based systems also exhibits many architectural styles. Each 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;

(4) **Semantic models**that enable a designer to understand the overall properties of a system by analysing the known properties of its constituent parts. In the section that follows, we consider commonly used architectural patterns for software.

3.6.1 A Brief Taxonomy of Styles and Patterns

a)Data-centered architectures.

- A data store (e.g., a file or database) resides at the centre of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 3.7 (Data-centered architecture) illustrates typical data-centred style. Client software accesses a central repository.

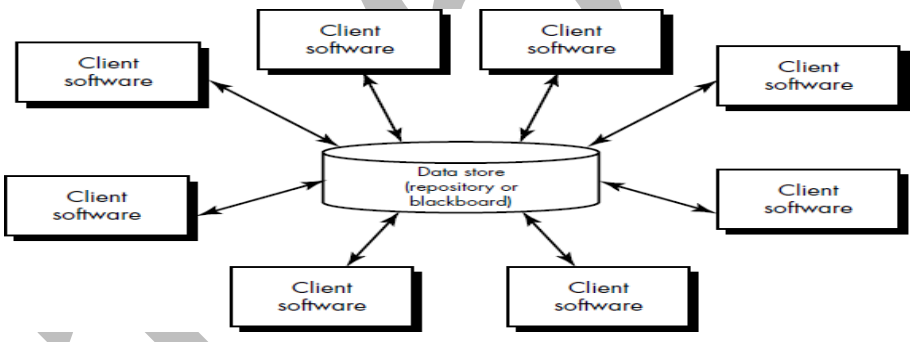


Figure 3.7 Data-centered architecture

- Data-centred architectures promote integrability. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients

b) 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.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighbouring filters.

- If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.

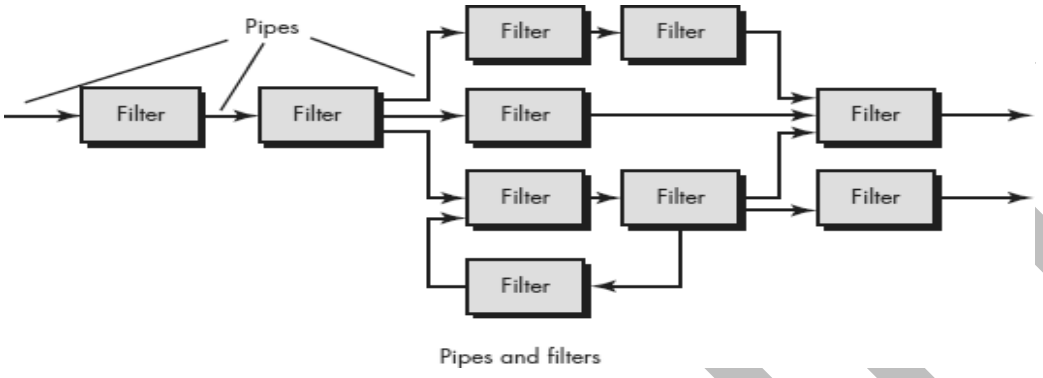


Figure 3.8 Data-flow architecture

c) Call and return architectures.

- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category:

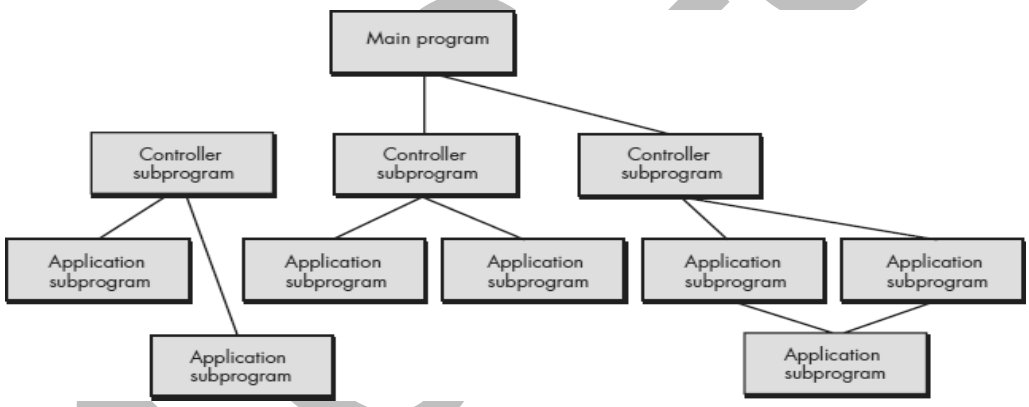


Figure 3.9 Main program/subprogram architecture

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

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

d)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 is accomplished via message passing.

e)Layered architectures.

- The basic structure of a layered architecture is illustrated in Figure 3.10. 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.
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen.

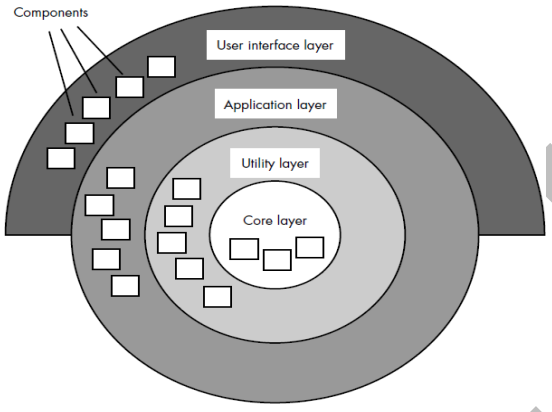


Figure 3.10 Layered architecture

3.6.2 Architectural Patterns

Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

3.6.3 Organization and Refinement

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 the architectural style that has been derived.

a) Control

- How is control managed within the architecture?
- Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
- How do components transfer control within the system?
- How is control shared among components?
- What is the control topology?
- Is control synchronized or do components operate asynchronously?

b) Data

- How are data communicated between components?
- Is the flow of data continuous, or are data objects passed to the system sporadically?
- What is the mode of data transfer?
- 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 ?

- How do data and control interact within the system?
- These questions provide the designer with an early assessment of design quality and lay the foundation for more-detailed analysis of the architecture.

3.7 ARCHITECTURAL DESIGN

- 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, and people) that the software interacts with and the nature of the interaction.
- Once context is modelled and all external software interfaces have been described, you can identify a set of architectural archetypes.
- An archetype is an abstraction (similar to a class) that represents one element of system behaviour.

3.7.1 Representing the System in Context

- Architectural context represents how the software interacts with entities external to its boundaries.
- At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries.
- The generic structure of the architectural context diagram is illustrated in Figure 3.11
- Referring to the figure 3.11, 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.

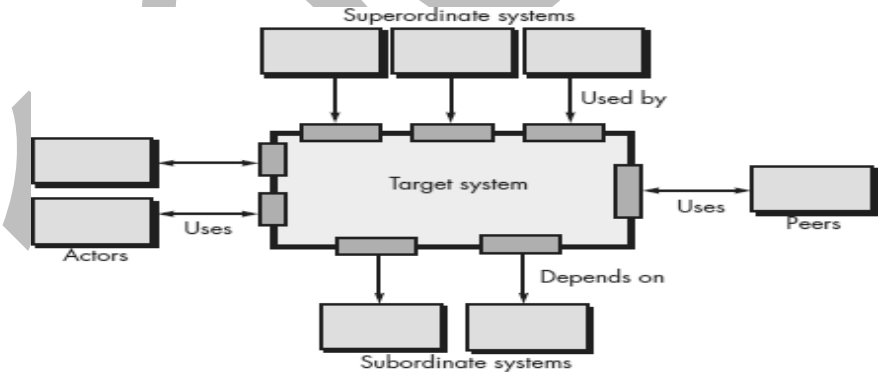


Figure 3.11 Architectural context diagram

- **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.

- To illustrate the use of the ACD, consider the home security function of the *SafeHome*product. The overall *SafeHome*product controller and the Internet-based system are both superordinate to the security function and are shown above the function in Figure 3.12.

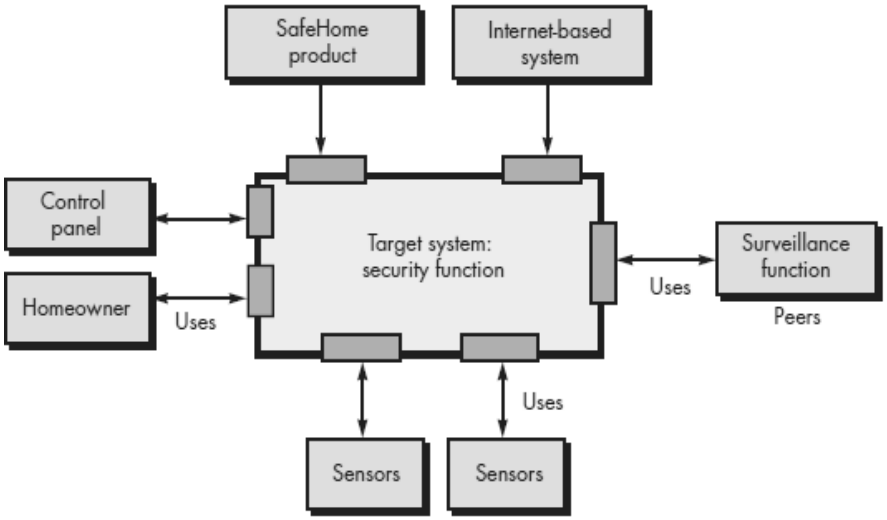


Figure 3.12 Architectural context diagram for the SafeHome security function

3.7.2 Defining Archetypes

- An archetype is a class or pattern that represents a core abstraction that is critical to the design of architecture for the target system.
- 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 behaviour of the system.
- The SafeHome home security function, defines 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 and (2) a variety of alarm (output) indicators.
- **Detector.**An abstraction that encompasses all sensing equipment that feeds information into the target system.
- **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.
- **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.

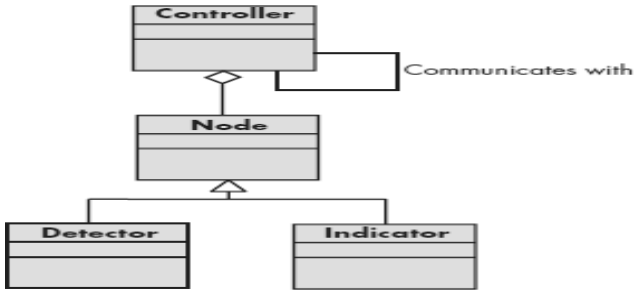


Figure 3.13 UML relationships for SafeHome security function archetypes

3.7.3 Refining the Architecture into Components

- As the software architecture is refined into components, the structure of the system begins to emerge.
- These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain.
  The components can be the entities that follow following functionalities.
- **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.

3.7.4 Describing Instantiations of the System

- The architectural design that has been modelled 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.

3.8 ARCHITECTURAL MAPPING USING DATA FLOW

- To illustrate one approach to architectural mapping, we consider the call and return Architecture—an extremely common structure for many types of systems.
- The mapping technique to be presented enables a designer to derive reasonably complex call and return architectures from data flow diagrams within the requirements model.
- The technique, sometimes called structured design,, top-down design , and structured programming.
- Structured design is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture.
- The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six-step process:

  (1) The type of information flow is established;
  (2) Flow boundaries are indicated;
  (3) The DFD is mapped into program structure;
  (4) Control hierarchy is defined;
  (5) Resultant structure is refined using design measures and heuristics;
  (6) The architectural description is refined and elaborated.

3.8.1 Transform Mapping

- Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.Consider the SafeHome security function.
- One element of the analysis model is a set of data flow diagrams that describe information flow within the security function.
- To map these data flow diagrams into a software architecture, the following design steps are:

Step 1. Review the fundamental system model.

- The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function.

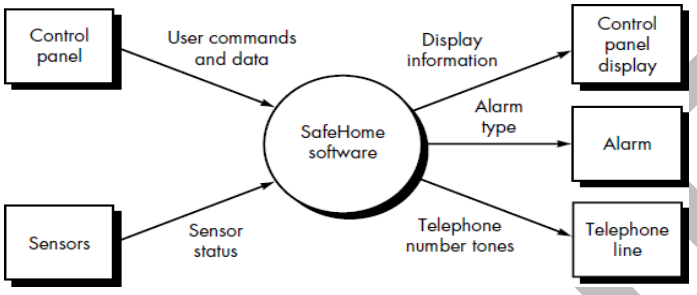


Figure 3.14 Context level 0 DFD for Safe Home

Step 2. Review and refine data flow diagrams for the software.

- Information obtained from analysis models contained in the Software Requirements Specifications refined to produce greater detail.
- For example, the level 2 DFD for monitor sensors is examined, and a level 3 data flow diagram is derived as, the process implied by a transform performs a single, distinct function that can be implemented as a module in the SafeHomesoftware.

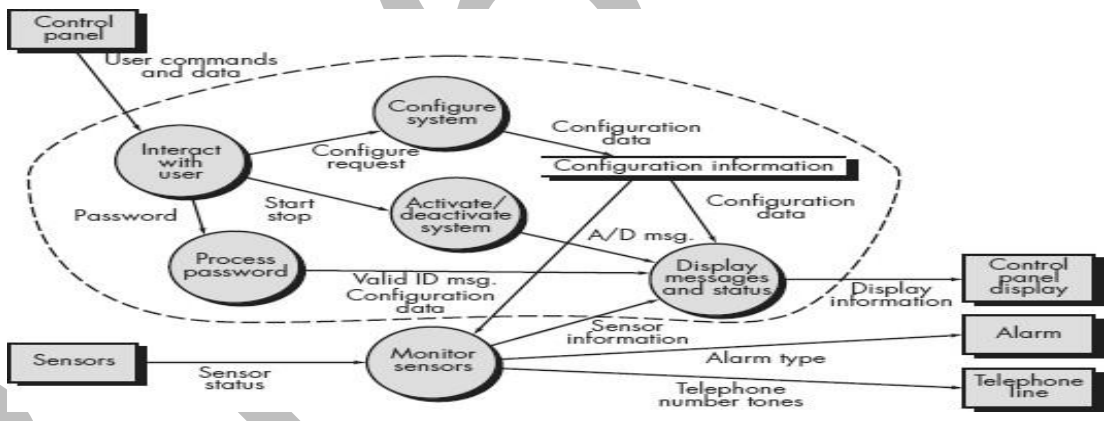


Figure3.15 Level 1 DFD for the SafeHome security function

- Therefore, the level 3 DFD contains sufficient detail for a "first cut" at the design of architecture for the monitor sensors subsystem, and we proceed without further refinement.

Step 3. Determine whether the DFD has transform or transaction flow characteristics.

- Evaluating the Level 3 DFD data entering the software along one incoming path and exiting along three outgoing paths.
- Therefore, an overall transform characteristic will be assumed for information flow.

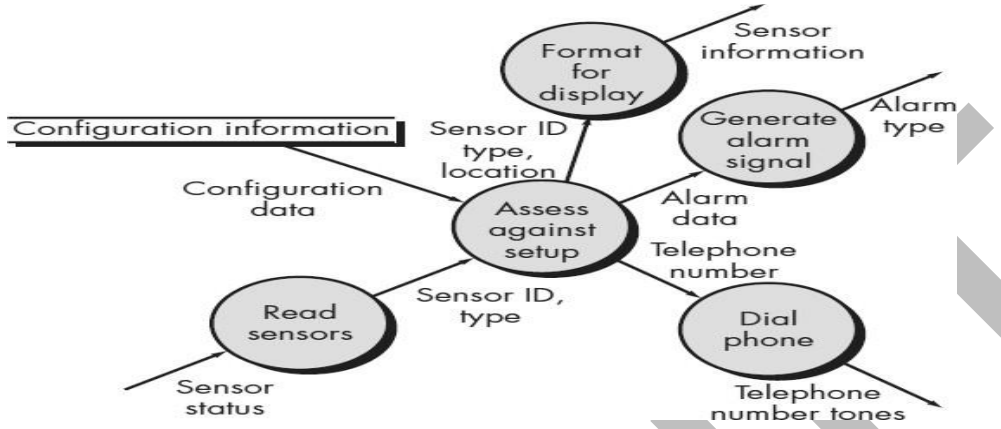


Figure 3.16 Level 2 DFD that refines the monitor sensors transform

Step 4. Isolate the transform centre by specifying incoming and outgoing flow boundaries

- Incoming flow as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form.

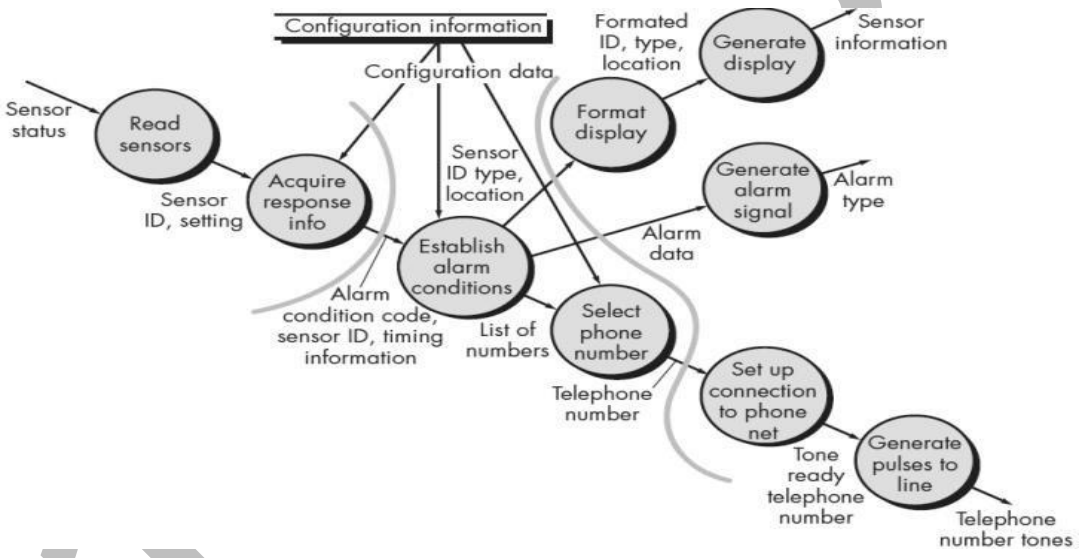


Figure 3.17 Level 3 DFD for monitor sensors with flow boundaries

- Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries.
- Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure. The transforms (bubbles) that constitute the transform centre lie within the two shaded boundaries that run from top to bottom in the figure.

Step 5. Perform "first-level factoring."

- When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing.

- An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.

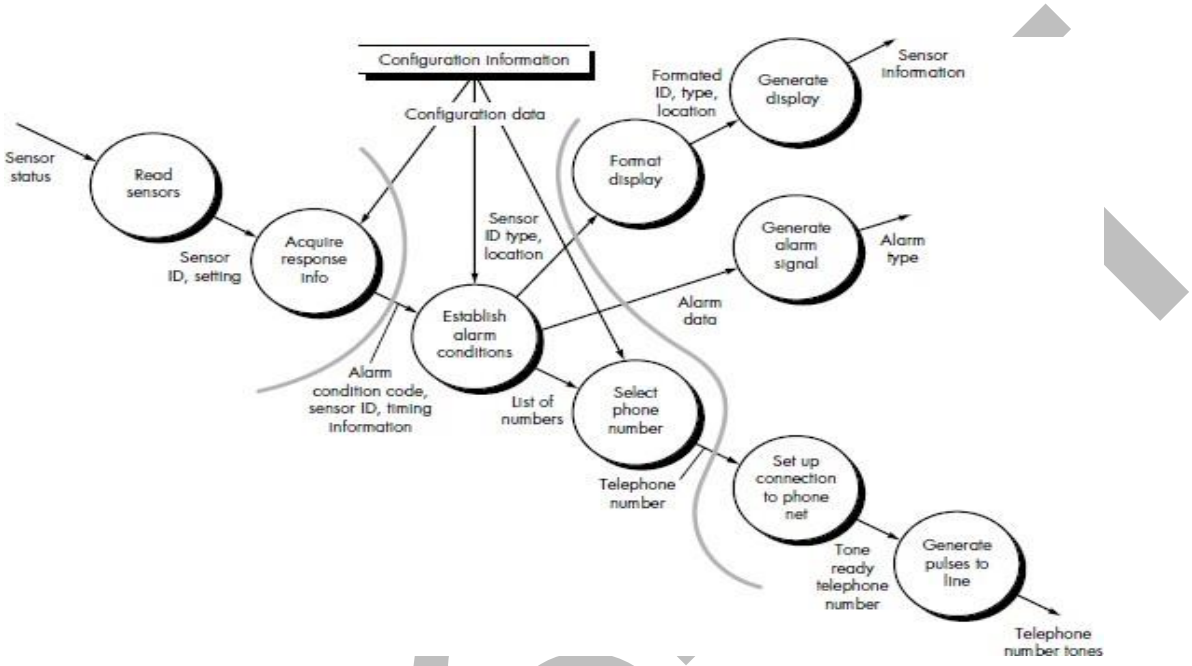


Figure 3.18 Level 3 DFD for monitor sensors with flow boundaries

- A transform flow controller, called alarm conditions controller, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).
- An outgoing information processing controller, called alarm output controller, coordinates production of output information.

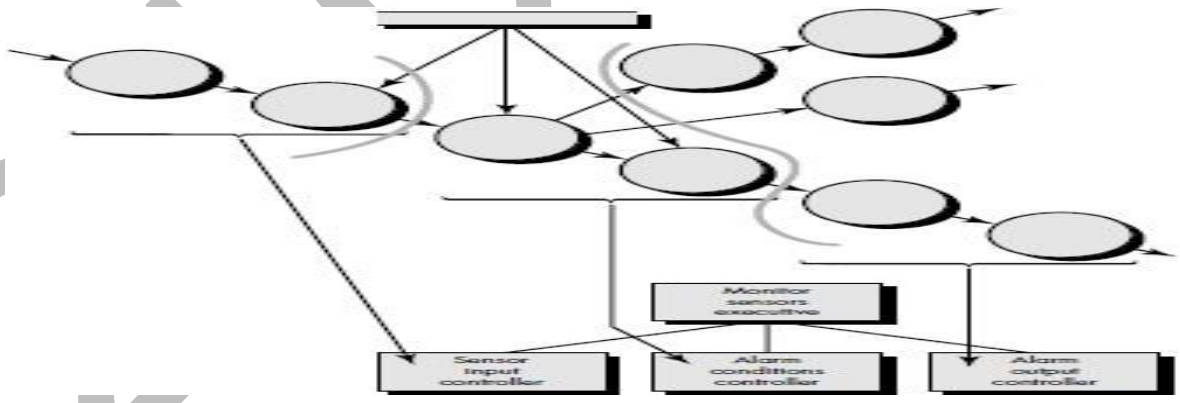


Figure 3.19 First-level factoring for monitor sensors

Step 6. Perform "second-level factoring."

- Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Second-level factoring for monitor sensors

- Although Figure 3.20 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur.
- Two or even three bubbles can be combined and represented as one module (recalling potential problems with cohesion) or a single bubble may be expanded to two or more modules.
- Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform centre boundary on the incoming flow side.

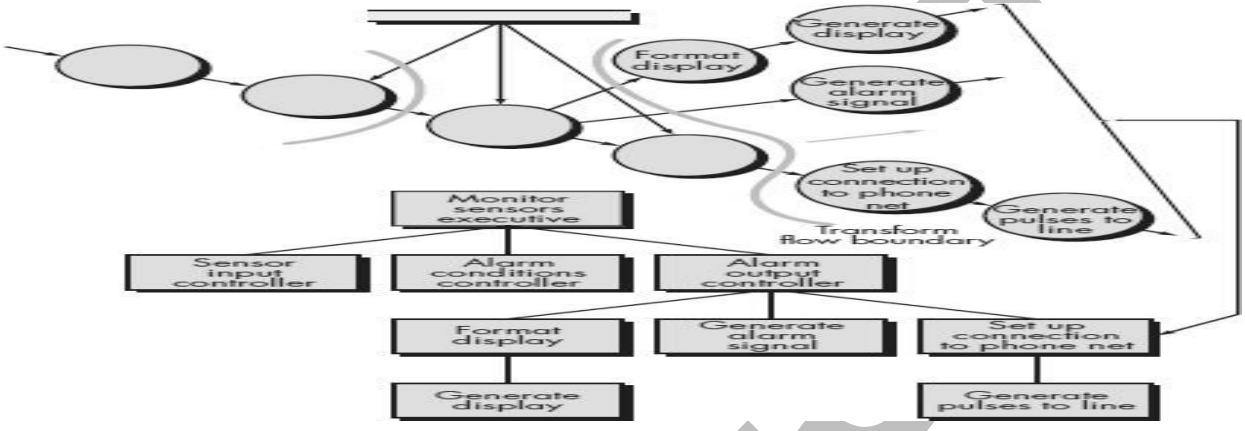


Figure 3.20 Second-layer Factoring

- The transform centre of monitor sensors subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first-iteration architecture is shown in Figure 3.21

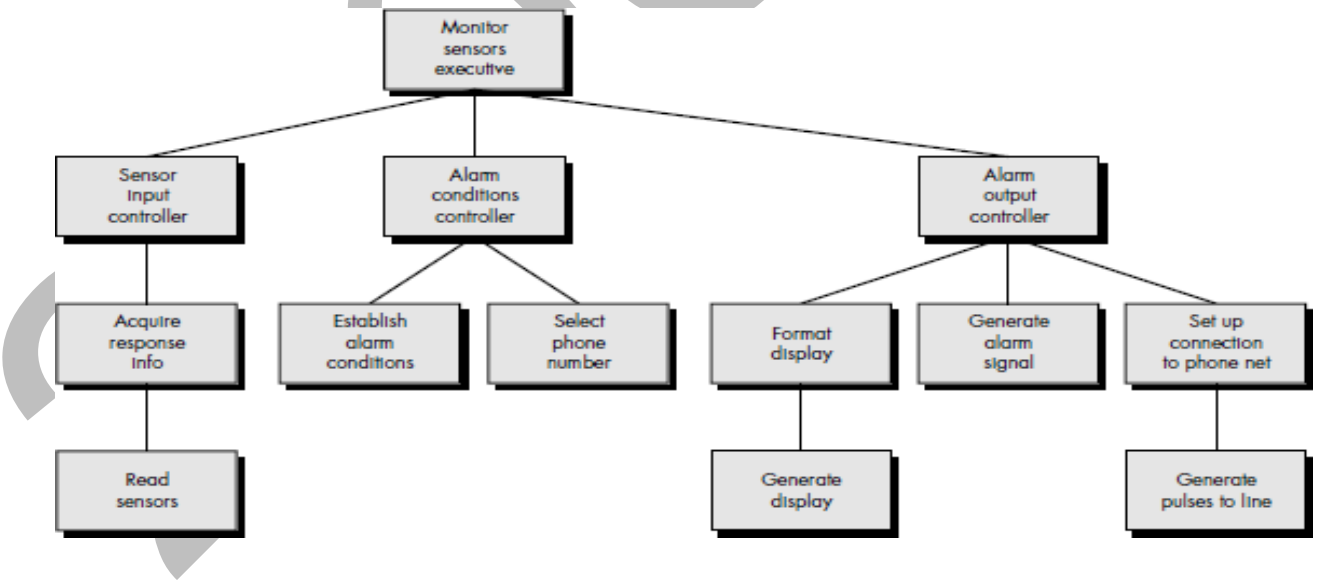


Figure 3.21 First-iteration program structure for monitor sensors

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. Many modifications can be made to the first iteration architecture developed for the Safe Home monitor sensors subsystem. Among many possibilities,

1. The incoming controller can be removed because it is unnecessary when a single incoming flow path is to be managed.

2. The substructure generated from the transform flow can be imploded into the module establish alarm conditions (which will now include the processing implied by select phone number). The transform controller will not be needed and the small decrease in cohesion is tolerable.

3. The modules format display and generate display can be imploded (we assume that display formatting is quite simple) into a new module called produce display.

- The refined software structure for the monitor sensors subsystem is shown in Figure 3.22.
- The objective of the preceding seven steps is to develop an architectural representation of software.

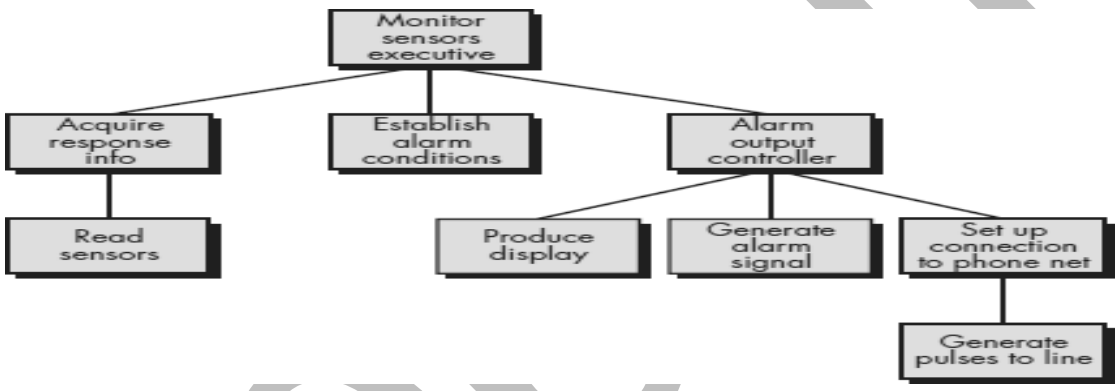


Figure 3.22 Refined program structure for monitor sensors

3.8.2 Transaction Mapping (Architectural Mapping)

- In many software applications, a single data item triggers one or a number of information
- flows that effect a function implied by the triggering data item.
- The data item, called a transaction.

An Example

- Transaction mapping will be illustrated by considering the user interaction subsystem of the SafeHomesoftware. Level 1 data flow for this subsystem is shown as part of Fig 3.15
- Refining the flow, a level 2 data flow diagram (a corresponding data dictionary, CSPEC, and PSPECs would also be created) is developed and shown in Figure 3.23
- As shown in the figure, **user commands** flows into the system and results in additional information flow along one of three action paths.
- A single data item, **command type,** causes the data flow to fan outward from a hub.
- It should be noted that information flow along two of the three action paths accommodateadditional incoming flow (e.g., system parameters and data are input on the "configure" action path). Each action path flows into a single transform, display messages and status.

Design Steps

- The design steps for transaction mapping are similar and in some cases identical to steps for transform mapping (Section 14.6).
- A major difference lies in the mapping of DFD to software structure.

Step 1. Review the fundamental system model.

Step 2. Review and refine data flow diagrams for the software.

Step 3. Determine whether the DFD has transform or transaction flow characteristics.

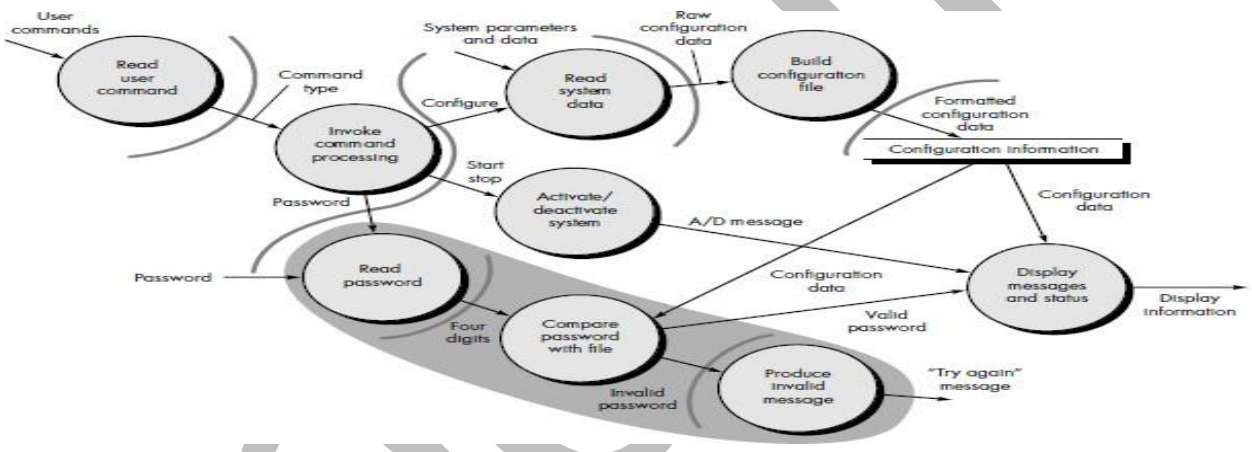


Figure 3.23Level 2 DFD for user interaction subsystem with flow boundaries

- Steps 1, 2, and 3 are identical to corresponding steps in transform mapping.
- The DFD shown in Figure 3.23 has a classic transaction flow characteristic.
- However, flow along two of the action paths emanating from the invoke command processing bubble appears to have transform flow characteristics. Therefore, flow boundaries must be established for both flow types.

Step 4. Identify the transaction centre and the flow characteristics along each of the action paths.

- The location of the transaction centre can be immediately discerned from the DFD.
- The transaction centre lies at the origin of a number of actions paths that flow radially from it. For the flow shown in Figure3.23, the invoke command processing bubble is the transaction centre.
- The incoming path (i.e., the flow path along which a transaction is received) and all action paths must also be isolated.
- Boundaries that define reception path and action paths are also shown in the figure 3.25. Each action path must be evaluated for its individual flow characteristic. For example, the "password"

path has transform characteristics. Incoming, transform, and outgoing flow are indicated with boundaries.

Step 5. Map the DFD in a program structure amenable to transaction processing.

- Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch.
- The structure of the incoming branch is developed in much the same way as transform mapping.
- Starting at the transaction centre, bubbles along the incoming path are mapped into modules. The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules.
- Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics. This process is illustrated schematically in Figure 3.23
- Considering the user interaction subsystem data flow, first-level factoring for step 5 is shown in Figure 3.25.
- The bubbles read user command and activate/deactivate system map directly into the architecture without the need for intermediate control modules.

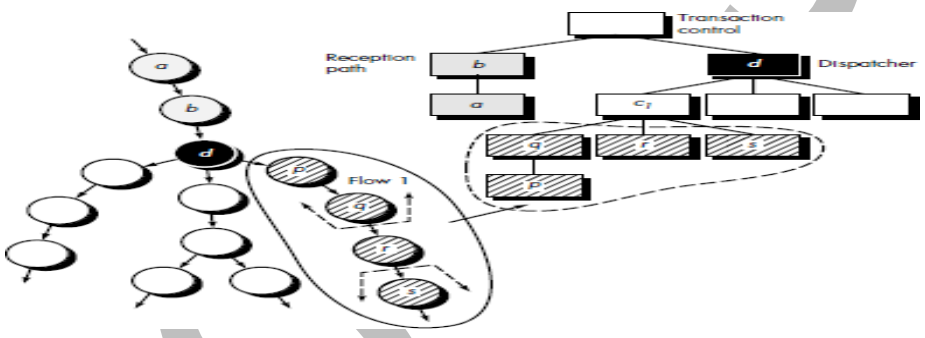


Figure 3.24Transaction mapping

- The transaction centre, invoke command processing, maps directly into dispatcher module of the same name. Controllers for system configuration and password processing are created as illustrated in Figure 3.23

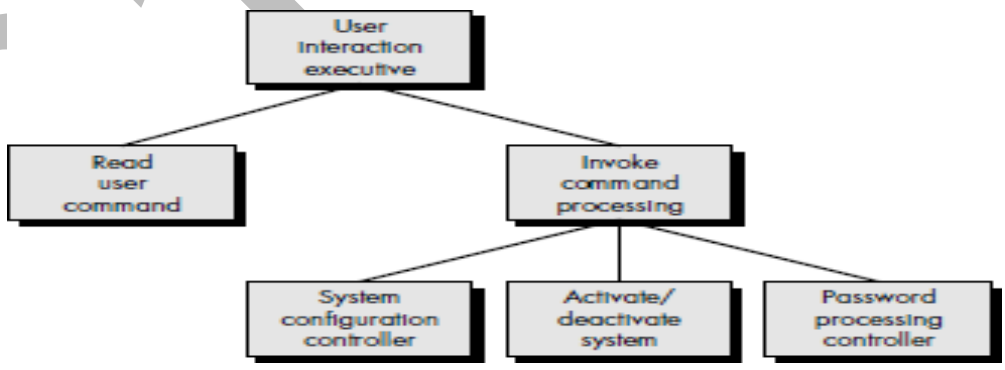


Figure 3.25First-level factoring for user interaction subsystem

Step 6. Factor and refine the transaction structure and the structure of each action path.
Each action path of the data flow diagram has its own information flow characteristics.

- We have already noted that transform or transaction flow may been countered. The action path-related "substructure" is developed using the design steps discussed in this and the preceding section.
- As an example, consider the password processing information flow shown (inside shaded area) in Figure3.23.
- The flow exhibits classic transform characteristics. A password is input (incoming flow) and transmitted to a transform center where it is compared against stored passwords.
- An alarm and warning message (outgoing flow)are produced (if a match is not obtained). The "configure" path is drawn similarly using the transform mapping. The resultant software architecture is shown in Figure 3.25

Step 7. Refine the first-iteration architecture using design heuristics forimproved software quality. This step for transaction mapping is identical to thecorresponding step for transform mapping. In both design approaches, criteria suchas module independence, practicality (efficacy of implementation and test), and maintainabilitymust be carefully considered as structural modifications are proposed.

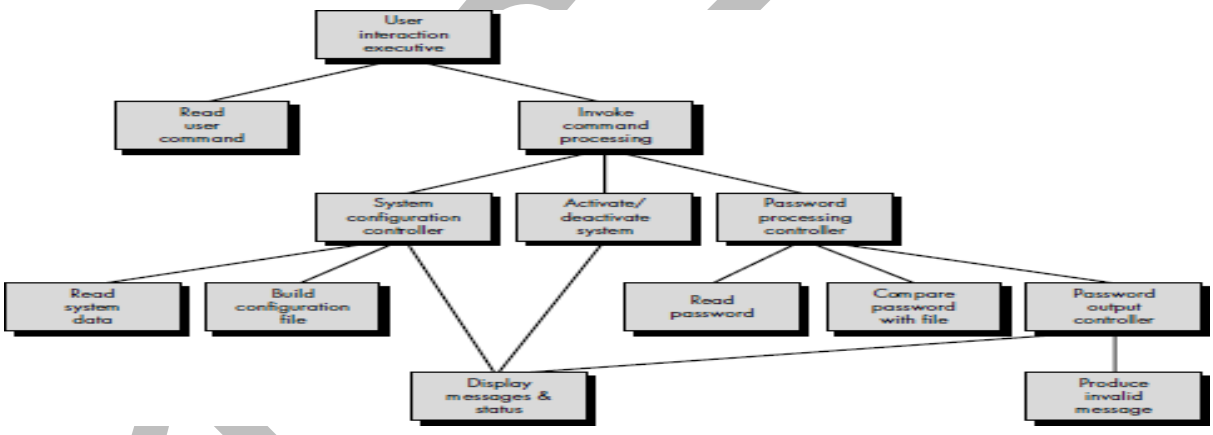


Figure 3.26 First-iteration architecture for user interaction subsystem

3.9 USER INTERFACE DESIGN

3.9.1 The Golden Rules

Mandel coins three golden rules:

1. Place the user in control.

2. Reduce the user's memory load.

3. Make the interface consistent.

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

1. **Place the user in control.**

Mandel defines a number of design principles that allow the user to maintain control:

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

2. Reduce the user's memory load.

Mandel defines design principles that enable an interface to reduce the user's memory load:

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real-world metaphor.
- Disclose information in a progressive fashion.

3. **Make the interface consistent.**

The interface should present and acquire information in a consistent fashion. This implies that

(1) all visual information is organized according to design rules that are maintained throughout all screen displays,

(2) input mechanisms are constrained to a limited set that is used consistently throughout the application

(3) mechanisms for navigating from task to task are consistently defined and implemented.

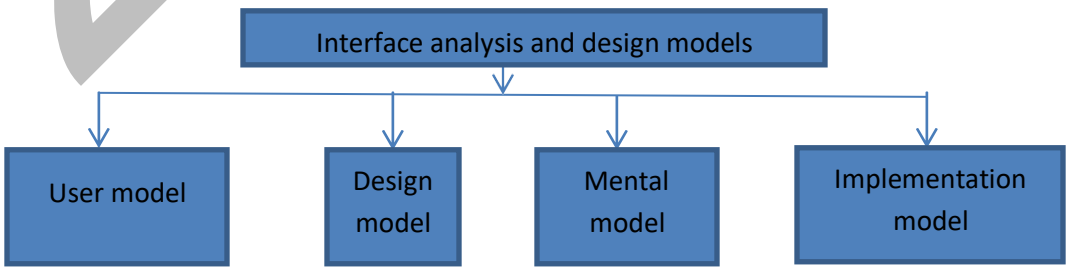
Mandel defines a set of design principles that help make the interface consistent:

- **Allow the user to put the current task into a meaningful context.**
- **Maintain consistency across a family of applications.**
- **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.**

3.10 USER INTERFACE ANALYSIS AND DESIGN

- The overall process for analyzing and designing a user interface begins with the creation of different models of system function.
- Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

3.10.1 Interface Analysis and Design Models



User model:

To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality"

In addition, users can be categorized as:

Novices. No syntactic knowledge1 of the system and little semantic knowledge2 of the application or computer usage in general.

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

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

Design model:

- It consists of data, architectural, interface and procedural representation of the software.

Mental model

- The user's mental model (system perception) is the image of the system that end users carry in their heads.

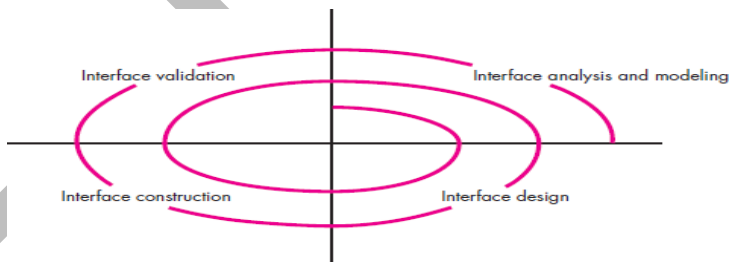
Implementation model:

- The implementation model combines the outward manifestation of the computer based system, coupled with all supporting information that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively.

3.10.2 The Process

- The analysis and design process for user interfaces is iterative and can be represented using a spiral model.
- The user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities

(1) Interface analysis and modeling,
(2) Interface design,
(3) Interface construction, and



(4) Interface validation.

Figure 3.27The user interface design process

- **Interface analysis** focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined.

- The **goal of interface design** is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.
- **Interface construction** normally begins with the creation of a prototype that enables usage scenarios to be evaluated.
- **Interface validation** focuses on

(1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;

(2) the degree to which the interface is easy to use and easy to learn, and

(3) the users' acceptance of the interface as a useful tool in their work.

3.11 INTERFACE ANALYSIS

- Before preceding for interface design it is necessary to understand the problem.
- Understanding the problem means understanding-

1. The people or user who actually interacts with the system.
2. The task that are performed by the end user for interacting the system.
3. The contents of the interface that will be displayed to the user.
4. The environment in which the task will be conducted.

3.11.1 User Analysis

- "User interface" is probably all the justification needed to spend some time understanding the user.
- Following are the ways by which one can learn **what the user wants from the user interface.**
- **User Interviews.** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and other issues. This can be accomplished in **one-on-one meetings** or through **focus groups.**
- **Sales input.** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.
- **Marketing input.** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.
- **Support input.** Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

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

- Are users trained professionals, technicians, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?

- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

3.11.2 Task Analysis and Modelling:

The goal of task analysis is to answer the following questions:

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?

In order to answer these questions following techniques are used-

Use cases.

- The use case describes the manner in which an actor interacts with a system.
- The use case is developed to show how an end user performs some specific work-related task.
- The use case is written in an informal style (a simple paragraph)

Task elaboration.

- **Task elaboration** also called functional decomposition or stepwise refinement as a mechanism for refining the processing tasks that are required for software to accomplish some desired function.
- Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.

Object elaboration.

- Rather than focusing on the tasks that a user must perform, examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer.
- These objects can be categorized into classes.
- **Attributes** of each class are defined, and an **evaluation** of the actions applied to each object provide a list of operations

Workflow analysis.

- When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis.
- This technique allows you to understand how a work process is completed when several people (and roles) are involved.
- Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram).
- Consider only a small part of the work process: the situation that occurs when a patient asks for a refill. Figure 3.28 presents a swimlane diagram that indicates the tasks and decisions for each of the three roles noted earlier.
- This information may have been elicited via interview or from use cases written by each actor. Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

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

2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., Fills prescription could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

Hierarchical representation.

- A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type.

User task: Requests that a prescription be refilled

- Provide identifying information.
- Specify name.
- Specify user id.
- Specify PIN and password.
- Specify prescription number.
- Specify date refill is required.

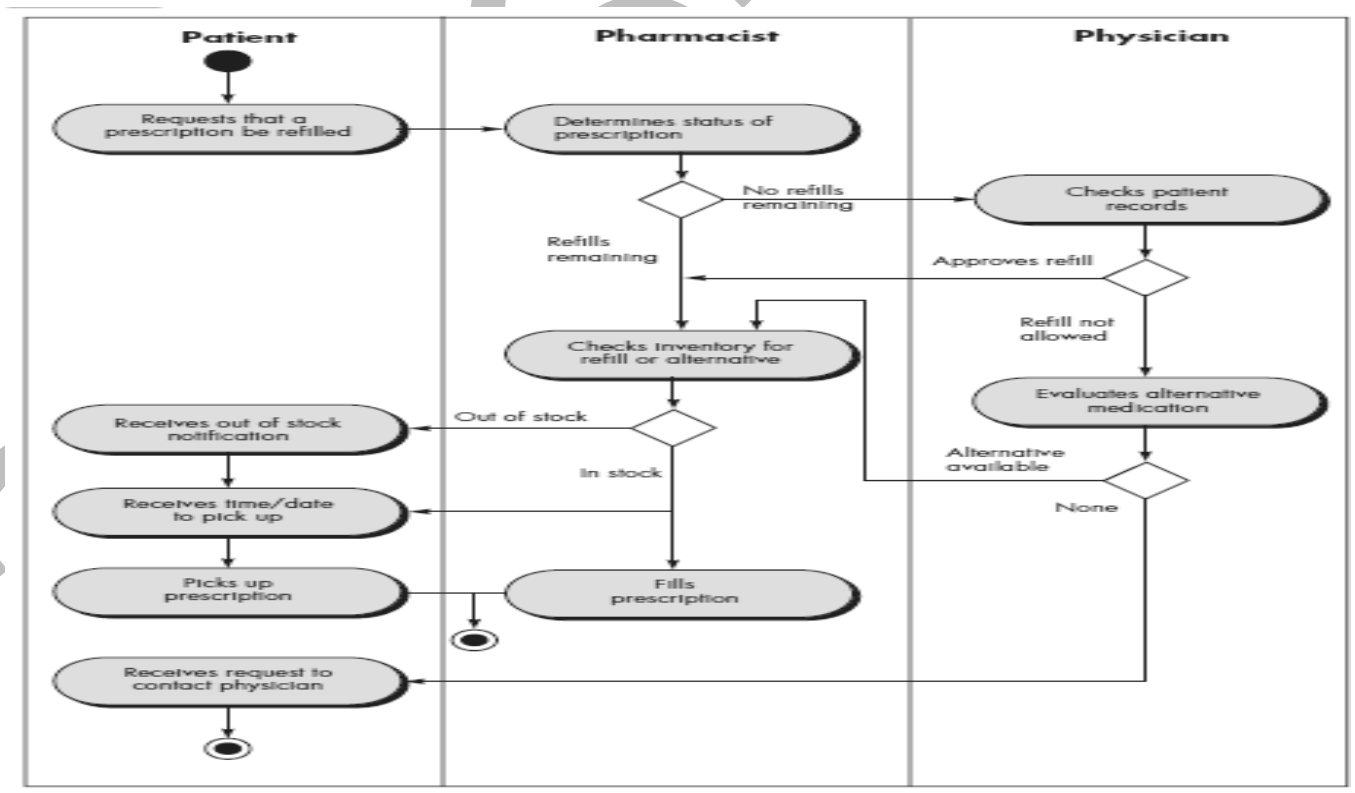


Figure 3.28 Swimlane diagram for prescription refill function

3.11.3 Analysis of Display Content

- The user tasks identified lead to the presentation of a variety of different types of content.
- For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files).
- During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered.
- Among the questions that are asked and answered are:
- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data?
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warnings be presented to the user?
- The answers to these (and other) questions will help you to establish requirements for content presentation.

3.11.4 Analysis of the Work Environment

- The analysis of work environment is very important
- In some applications the user interface for a computer-based system is placed in a "user- friendly location" (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.
- In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

3.12 INTERFACE DESIGN STEPS

- Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.
- Following are the commonly used interface design steps-

1. Using information developed during interface analysis define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.

4. Indicate how the user interprets the state of the system from information provided through the interface.

While designing the interface the designer has to follow-

- Golden rules
- Model the interface
- Analyze the working environment

3.12.1 Applying Interface Design Steps

- The first step in any interface design is to identify all the necessary objects and corresponding actions.
- There are three types of objects
- Target,
- source, and
- application objects
- A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon).
- An **application object** represents application-specific data that are not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing.

3.12.2 User Interface Design Patterns

- Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged.
- A design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

3.12.3 Design Issues

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

Response time:

- system response time is measured from the point at which the user performs some control action.

System response time Characteristics	System length	The amount of time taken by the system to respond
	Variability	Variation of time from average response time

Help facilities

- This is the most essential criteria for any user interface this makes the system more interactive.
- The help can be online help or it can be in the form of user manual.

Error handling

- Errors and warning cause the frustrations to the user.

Menu and command labelling

- The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type.

- There are number of design issues for typed command and menu labels.

Command related design issues	**Menu label related design issues**
What form will commands take?	Will every menu option have a corresponding command?
How difficult will it be to learn and remember the commands?	Are menu labels self-explanatory within the context of the interface?
Can commands be customized or abbreviated by the user?	Are submenus consistent with the function implied by a master menu item?
What can be done if a command is forgotten?	

Application accessibility

- Accessibility for users and software engineers may be physically challenged is an imperative for ethical, legal, and business reasons.
- A variety of accessibility guidelines designed for Web applications but often applicable to all types of software—provides detailed suggestions for designing interfaces that achieve varying levels of accessibility.

Internationalization

- Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages.
- A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements).
- The Unicode standard has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

3.13 COMPONENT LEVEL DESIGN:

- A component is a modular building block for computer software.
- The OMG Unified Modeling Language Specification defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

3.14 DESIGNING CLASS-BASED COMPONENTS

- Component is represented as part of the architectural model.
- When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model.
- The detailed description of the attributes, operations, and interfaces used by these classes.

3.14.1 Basic Design Principles

There are four design principles that are used during the component level design. These principles are-

The Open-Closed Principle (OCP)

- "A module [component] should be open for extension but closed for modification"

- A designer should design a component in such a manner that some functionalities can added to it if required but in doing so there should not be any change in the internal design of the component itself.

The Liskov Substitution Principle (LSP)

- "Subclasses should be substitutable for their base classes"
- A component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.

Dependency Inversion Principle (DIP)

- "Depend on abstractions. Do not depend on concretions"
- The more a component depends on other concrete components rather than on abstractions such as an interface, the more difficult it will be to extend.

The Interface Segregation Principle (ISP)

- "Many client-specific interfaces are better than one general purpose interface"
- There are many instances in which multiple client components use the operations provided by a server class.

The Release Reuse Equivalency Principle (REP)

- "The granule of reuse is the granule of release".
- When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it.
- The developer commits to establish a release control system that supports and maintains older versions of the entity while the users slowly upgrade to the most current version.

The Common Closure Principle (CCP)

- "Classes that change together belong together."
- Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area.
- When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.

f)The Common Reuse Principle (CRP).

- "Classes that aren't reused together should not be grouped together".
- When one or more classes within a package changes, the release number of the package changes.
- All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested to ensure that the new release operates without incident.

3.14.2 Component-Level Design Guidelines

- Ambler suggests the following guidelines for conducting component level design
- **Components:** Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.
- **Interfaces:**Interfaces provide important information about communication and collaboration (as well as helping us to achieve the OCP).

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

3.14.3 Cohesion:

- Component-level design for object-oriented systems, cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

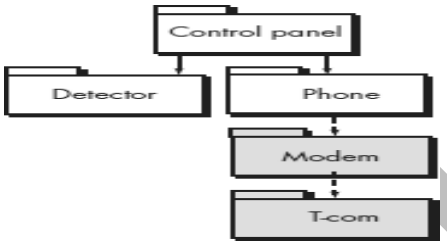


Figure 3.29 Layer cohesion

Various types of cohesion are:

Functional.

- This level of cohesion occurs when a component performs a targeted computation and then returns a result.

Layer.

- Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

Communicational.

- All operations that access the same data are defined within one class.

3.14.4 Coupling

- Coupling is a qualitative measure of the degree to which classes are connected to one another.
- Various types of coupling are
- **Content coupling.** Occurs when one component "surreptitiously modifies data that is internal to another component"
- **Common coupling.**Occurs when a number of components all make use of a global variable.
- **Control coupling.** Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B.
- **Stamp coupling.**Occurs when ClassB is declared as a type for an argument of an operation of Class A. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.
- **Data coupling.**Occurs when operations pass long strings of data arguments.
- **Routine call coupling.**Occurs when one operation invokes another.
- **Type use coupling.** Occurs when component A uses a data type defined in component B.If the type definition changes, every component that uses the definition must also change.
- **Inclusion or import coupling.**Occurs when component A imports or includes a package or the content of component B.

- **External coupling.** Occurs when a component communicates or collaborates with infrastructure components

3.15 TRADITIONAL COMPONENTS

- Component level design is also called as **procedural design.**
- The goal of component level design is to translate design model into operational software.
- Graphical, tabular or text based notations are used to create a set of **structured programming constructs.**
- The constructs are **sequence, condition, and repetition.**
- **Sequence** implements processing steps that are essential in the specification of any algorithm.
- **Condition** provides the facility for selected processing based on some logical occurrence.
- **Repetition** allows for looping.

3.15.1 Graphical Design Notation:

- The structured programming constructs can be represented by graphical notations. These graphical notations are called flow chart.
- A box is used to indicate a processing step.
- A diamond represents a logical condition, and arrows show the flow of control. Figure illustrates three structured constructs.

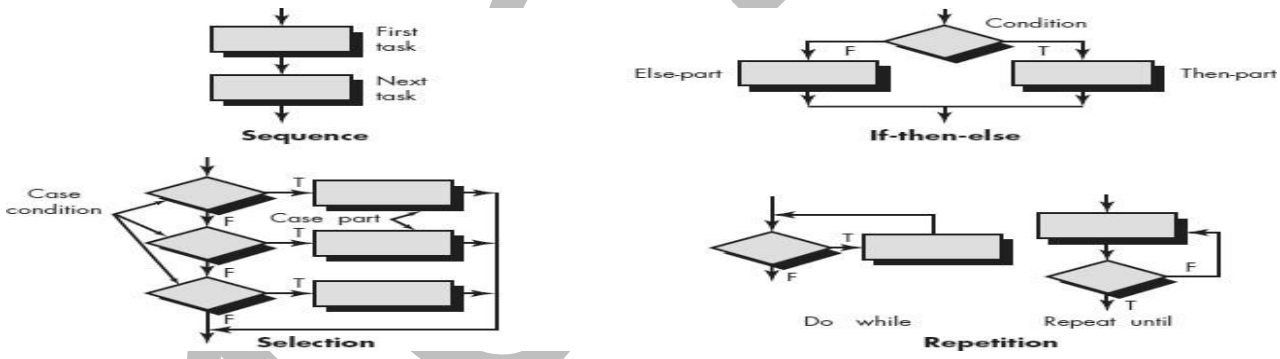


Figure 3.30 Flowchart constructs

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

3.15.2 Tabular Design Notation

- Decision tables provide a notation that translates actions and conditions described in a processing narrative or a use case into a tabular form.
- The table 4.1 is divided into four sections.
- The upper left-hand quadrant contains a list of all conditions.
- The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions.

- The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination.
- Therefore, each column of the matrix may be interpreted as a processing rule.
- The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or component).
2. List all conditions (or decisions made) during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what actions occur for a set of conditions.

	Rules					
Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

Table 3.1 Decision table nomenclature

3.15.3 Program Design Language

- Program design language (PDL), also called structured English or pseudocode, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language.
- A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs.
- The following PDL provides an elaboration of the procedural design for an early version of an alarm management component.

```
component alarmManagement;
The intent of this component is to manage control panel switches and input from sensors by
type and to act on any alarm condition that is encountered.
    set default values for systemStatus (returned value), all data items
    initialize all system ports and reset all hardware
    check controlPanelSwitches (cps)
        if cps = "test" then invoke alarm set to "on"
        if cps = "alarmOff" then invoke alarm set to "off"
        if cps = "newBoundingValue" then invoke keyboardInput
        if cps = "burglarAlarmOff" invoke deactivateAlarm;
        •
        •
        •
```

```
        default for cps = none
    reset all signalValues and switches
    do for all sensors
        invoke checkSensor procedure returning signalValue
        if signalValue > bound [alarmType]
            then phoneMessage = message [alarmType]
                set alarmBell to "on" for alarmTimeSeconds
                set system status = "alarmCondition"
                parbegin
                    invoke alarm procedure with "on", alarmTimeSeconds;
                    invoke phone procedure set to alarmType, phoneNumber
                endpar
            else skip
        endif
    enddofor
end alarmManagement
```