Architectural design

- Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements.
- Architectural design represents the structure of data and program components that are required to build a computer-based system. Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system.
- 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. Analyze the effectiveness of the design in meeting its stated requirements.
- 2. Consider architectural alternatives at a stage when making design changes is still relatively easy.
- 3. Reducing the risks associated with the construction of the software.
- Software architecture considers two levels of the design Pyramid:
- 1. Data design: data design enables us to represent the data component of the architecture.
- 2. Architectural design: Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Data design

- Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction (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. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.
- 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 (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

Data Modeling, Data Structures, Databases, and the Data Warehouse

- The data objects defined during software requirements analysis are modeled using entity/relationship diagrams and the data dictionary . The data design activity translates these elements of the requirements model into data structures at the software component level and, when necessary, a database architecture at the application level.
- In past year, data architecture was generally limited to data structures at the program level and databases at the application level. But today, businesses large and small are awash in data. It is not unusual for even a moderately sized business to have dozens of databases serving many applications encompassing hundreds of gigabytes of data. The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is crossfunctional (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

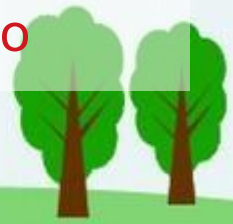
- To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in databases (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment. An alternative solution, called a data warehouse, adds an additional layer to the data architecture.
- A data warehouse is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business . In a sense, a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business. But many characteristics differentiate a data warehouse from the typical database :

- **Subject orientation**. Data in a warehouse is organized around major business subjects (e.g., customers, products, sales) rather than specific business processes or applications, making it easier to analyze information across different departments
- **Integration.** Data is collected from various disparate source systems and brought together into a consistent format and structure within the data warehouse, resolving inconsistencies and ensuring data integrity
- **Time variancy.** The warehouse maintains a historical record of data over long periods. Unlike operational databases that typically store only the current state of data, a data warehouse allows for the analysis of trends and performance over time. For a transaction-oriented application environment, data are accurate at the moment of access and for a relatively short time span (typically 60 to 90 days) before access. For a data warehouse, however, data can be accessed at a specific moment in time (e.g., customers contacted on the date that a new product was announced to the trade press). The typical time horizon for a data warehouse is five to ten years.
- **Non-volatility.** Data in a data warehouse is stable; once data is stored, it is generally not updated or deleted in real-time. New data is typically appended incrementally. Unlike typical business application databases that undergo a continuing stream of changes (inserts, deletes, updates), data are loaded into the warehouse, but after the original transfer, the data do not change.

Data Design at the Component Level

- Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. Wasserman has proposed a set of principles that may be used to specify and design such data structures. In actuality, the design of data begins during the creation of the analysis model. Recalling that requirements analysis and design often overlap, we consider the following set of principles for data specification:

1. The systematic analysis principles applied to function and behavior should also be applied to data. We spend much time and effort deriving, reviewing, and specifying functional requirements and preliminary design. **Representations of data flow and content should also be developed and reviewed, data objects should be identified, alternative data organizations should be considered, and the impact of data modeling on software design should be evaluated.** For example, specification of a multiringed linked list may nicely satisfy data requirements but lead to an unwieldy software design. An alternative data organization may lead to better results.



2. **All data structures and the operations to be performed on each should be identified.** The design of an efficient data structure must take the operations to be performed on the data structure into account . For example, consider a data structure made up of a set of diverse data elements. The data structure is to be manipulated in a number of major software functions. Upon evaluation of the operations performed on the data structure, an abstract data type is defined for use in subsequent software design. Specification of the abstract data type may simplify software design considerably.

3. **A data dictionary should be established and used to define both data and program design.** The concept of a data dictionary has been introduced in . A data dictionary explicitly represents the relationships among data objects and the constraints on the elements of a data structure. Algorithms that must take advantage of specific relationships can be more easily defined if a dictionary like data specification exists.

4. Low-level data design decisions should be deferred until late in the design process. **A process of stepwise refinement may be used for the design of data.** That is, overall data organization may be defined during requirements analysis, refined during data design work, and specified in detail during component level design. The top-down approach to data design provides benefits that are analogous to a top-down approach to software design—major structural attributes are designed and evaluated first so that the architecture of the data may be established.

5. **The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.** The concept of information hiding and the related concept of coupling provide important insight into the quality of a software design. This principle alludes to the importance of these concepts as well as "the importance of separating the logical view of a data object from its physical view".

6. A library of useful data structures and the operations that may be applied to them should be developed. Data structures and operations should be viewed as a resource for software design. Data structures can be designed for reusability. A library of data structure templates (abstract data types) can reduce both specification and design effort for data.

7. A software design and programming language should support the specification and realization of abstract data types. The implementation of a sophisticated data structure can be made exceedingly difficult if no means for direct specification of the structure exists in the programming language chosen for implementation.

Architecture Design

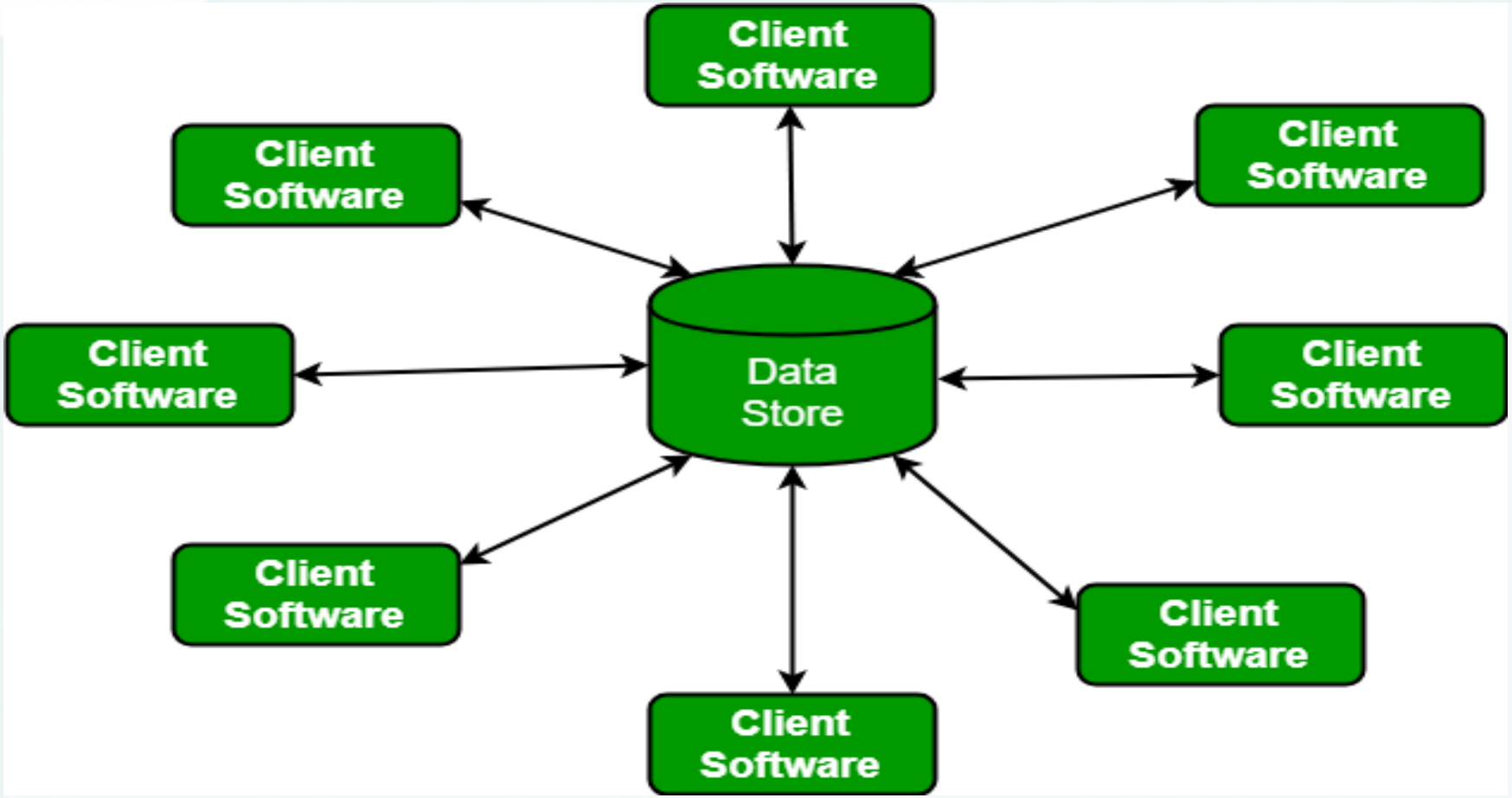
- The software needs the architectural design to represents the design of software. IEEE defines architectural design as “the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.” The software that is built for computer-based systems can exhibit one of these many architectural styles.
- Each style will describe a system category that consists of :
- **A set of components**(eg: a database, computational modules) that will perform a function required by the system.
- **The set of connectors** will help in coordination, communication, and cooperation between the components.
- **Conditions** that how components can be integrated to form the system.
- **Semantic models** that help the designer to understand the overall properties of the system.

The use of architectural styles is to establish a structure for all the components of the system.

Taxonomy of Architectural styles:

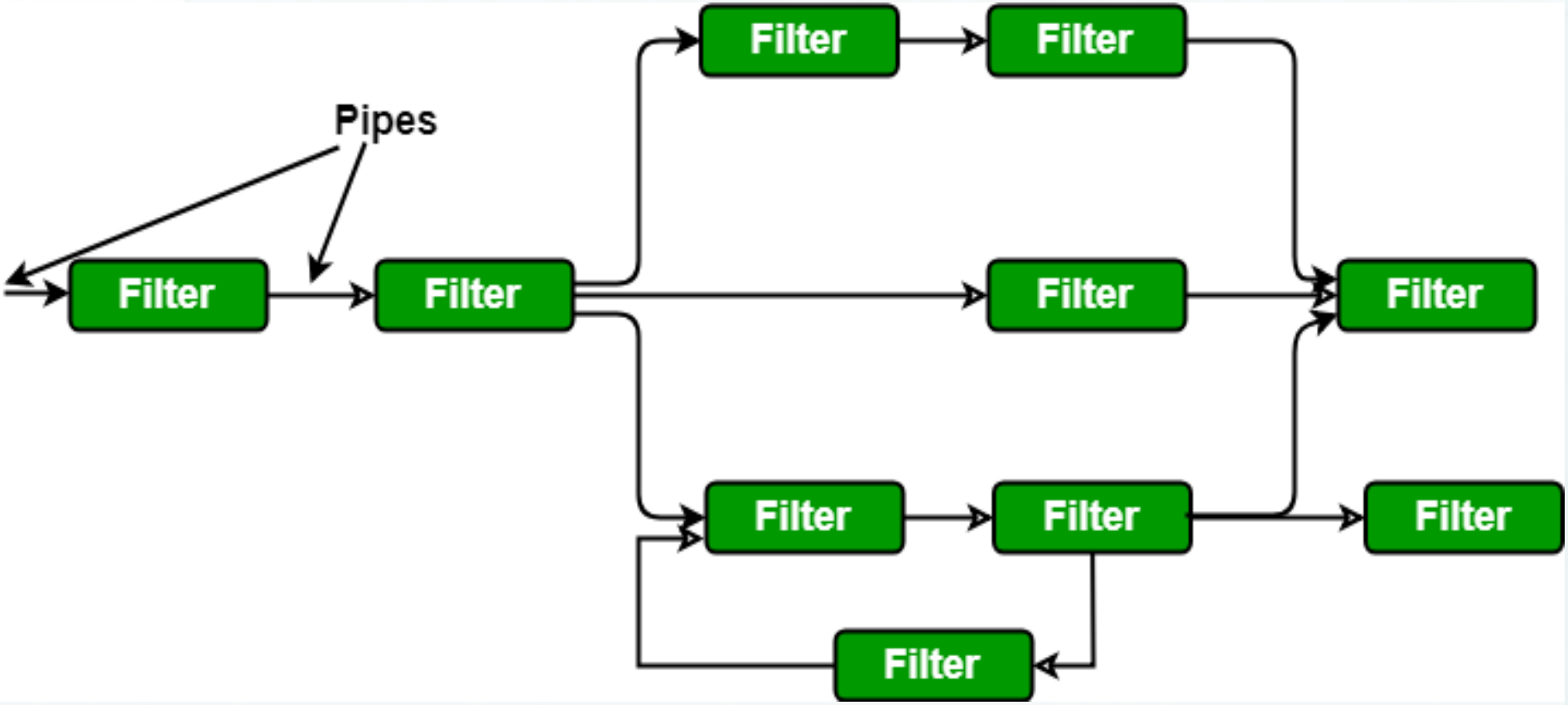
1. Data centered architectures:

- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
- The figure illustrates a typical data centered style. The client software access a central repository. Variation of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
- Data can be passed among clients using blackboard mechanism (notification).


Client Software
Client Software
Client Software
Client Software
Data Store
Client Software
Client Software
Client Software
Client Software

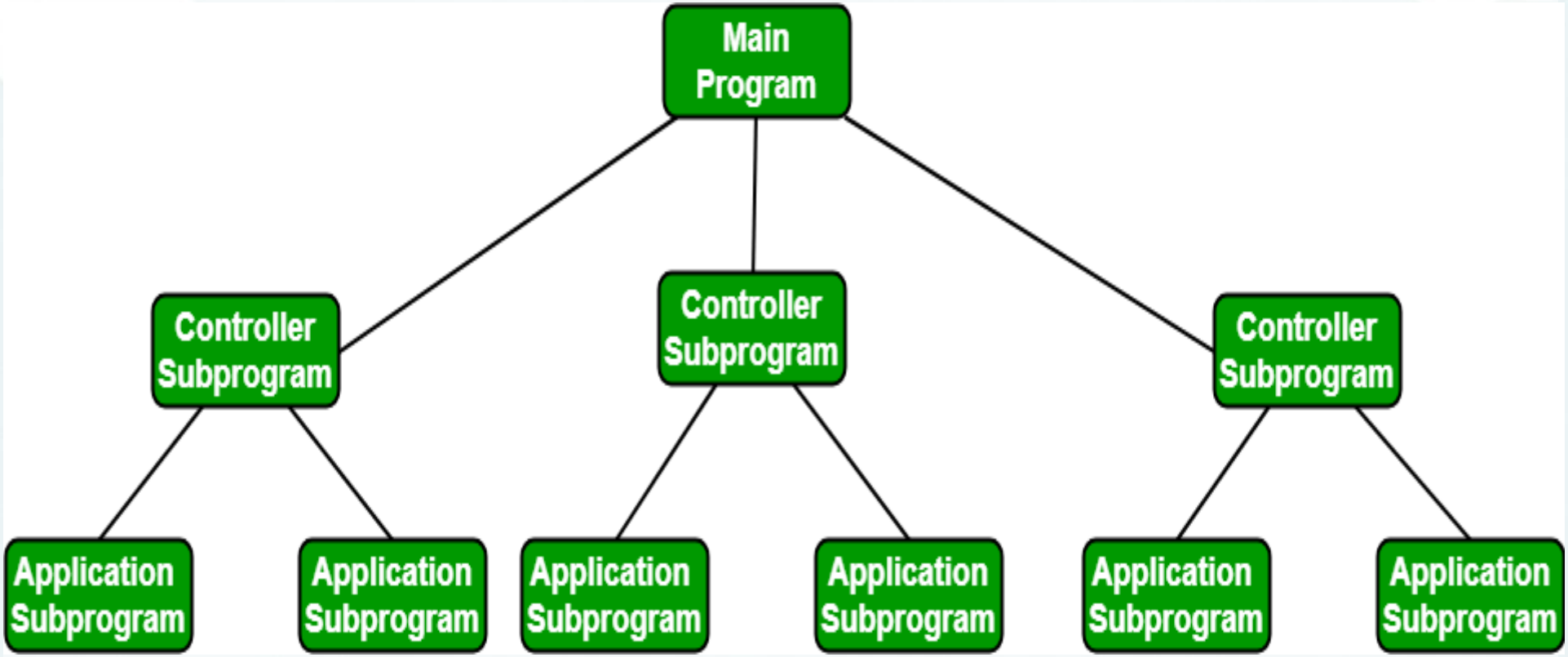
2.Data flow architectures:

- This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.
- Pipes are used to transmit data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don’t require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.


Pipes
Filter
Filter
Filter
Filter
Filter
Filter
Filter
Filter
Filter
Filter

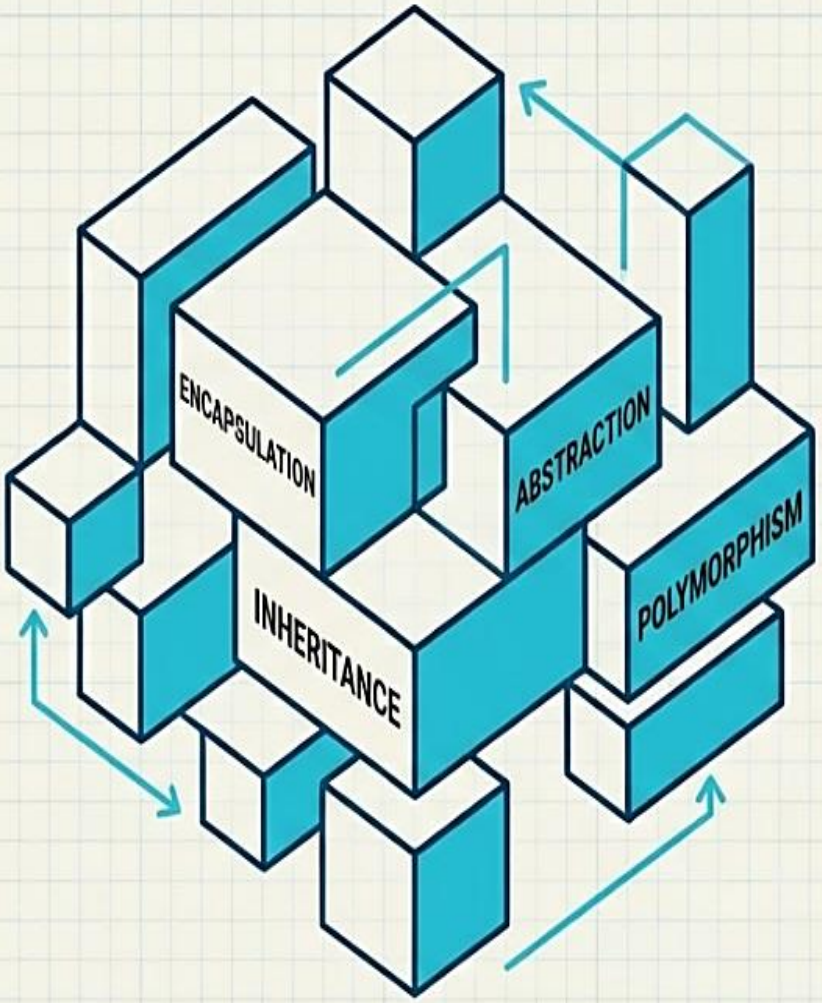
3.Call and Return architectures:

- It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.
- Remote procedure call architecture: This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- Main program or Subprogram architectures: The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.


Main Program
Controller Subprogram
Controller Subprogram
Controller Subprogram
Application Subprogram
Application Subprogram
Application Subprogram
Application Subprogram
Application Subprogram
Application Subprogram

4.Object Oriented architecture:

- The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.



Definition:
The system is constructed from a collection of interacting objects, which are instances of classes. This is the base of most modern software applications.

Core Concepts:

- **Object**: An instance of a class (e.g., Student, Company).
- **Class**: A collection of attributes and methods.
- **Encapsulation**: Binding similar types of data and elements together.
- **Abstraction**: Hiding irrelevant details to show only required information.
- **Inheritance**: Creating parent-child relationships between classes.
- **Polymorphism**: Allowing for the generation of multiple forms from a single entity.

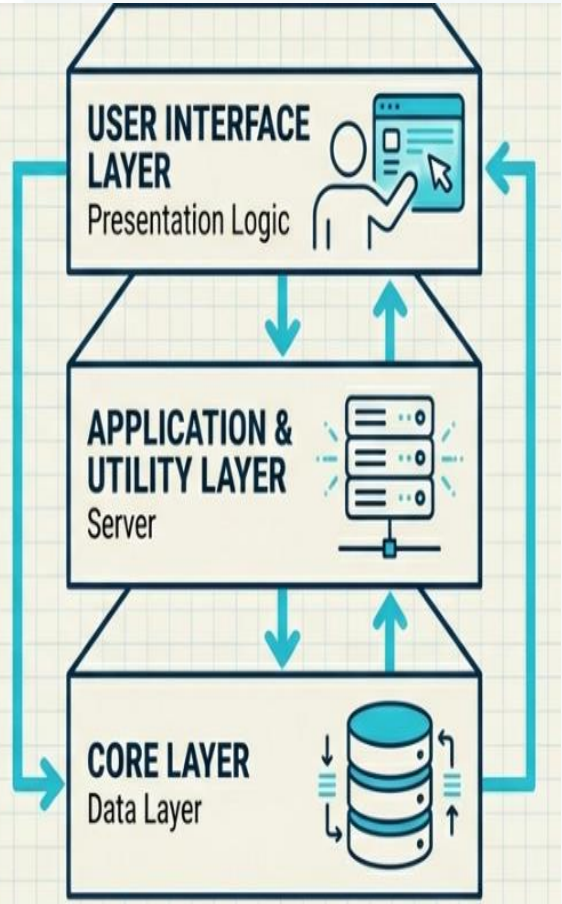
5.Layered architecture:

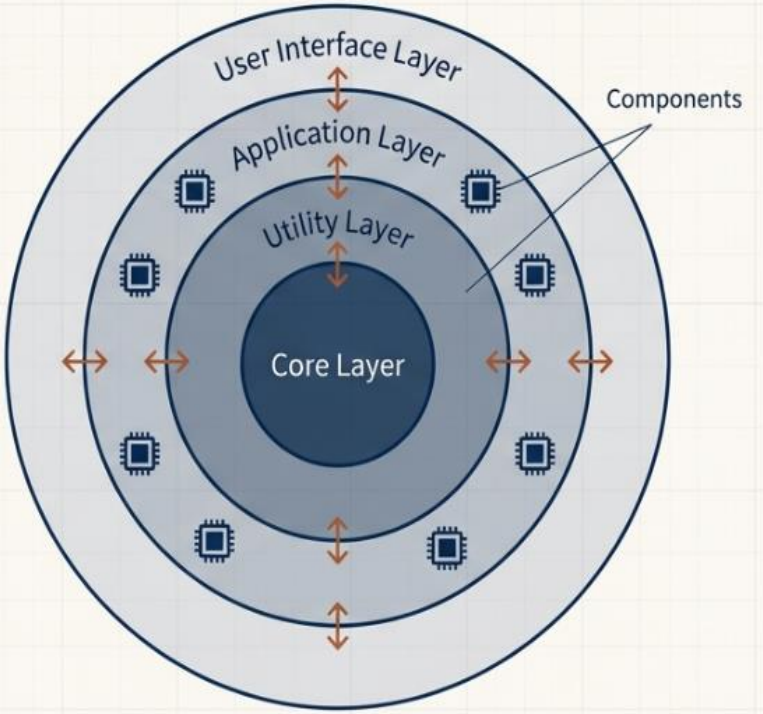
Definition:
Data moves from one level to another for processing, with each layer having a specific responsibility and providing services to the layer above it.

Real-World Example:
E-commerce web applications like Amazon or Flipkart.

Key Layers:

- **User Interface Layer (Outer):** What the user interacts with (front-end).
- **Application & Utility Layer (Intermediate):** Server-side logic (HTTP requests, etc.).
- **Core Layer (Inner):** Database and core programming logic.





Example: E-commerce applications like Amazon or Flipkart.

Typical Layers:

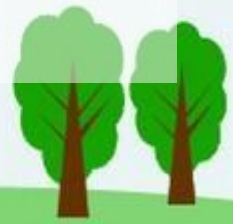
- **User Interface Layer (Outer)**: What the user sees and interacts with.
- **Application Layer**: Implements business logic and services.
- **Utility Layer**: Provides common, reusable functions.
- **Core Layer (Inner)**: Handles low-level functions like OS and database communication.

- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural styles and or combination of patterns that best fits those characteristics and constraints can be chosen.
- In many cases more than one pattern might be appropriate and alternative architectural styles can be designed and evaluated. For example a layered style can be combined with a data centered architecture in many database applications.
- Choosing the right architecture style can be tricky.
- Problem frames describe characteristics of recurring problems, without being distracted by references to details of domain knowledge or programming solution implementations.
- Domain driven design suggests that the software design should reflect the domain and the domain logic of the business problem we want to solve with our application.

- A problem frame is a generalization of a class of problems that might be used to solve the problem at hand. There are five fundamental problem frames and these are often associated with architectural styles:
- Simple work pieces(tools), required behavior (data centered), commanded behavior(command processor), information display(observer), and transformation(pipe and filter variants)
- Real world problem often follow more than one problem frame and as a consequence an architectural model may be combination of different frames example the model-view-controller(MVC) architecture used in WebAPP design might be viewed as combining two problem frames(command behavior and information display).in MVC the end user command is sent from the browser window to command processor(controller) which manages access to the content(model) and instructs the information rendering model(view) to translate it for display by the browser software.

Mapping Requirements in Software Architecture

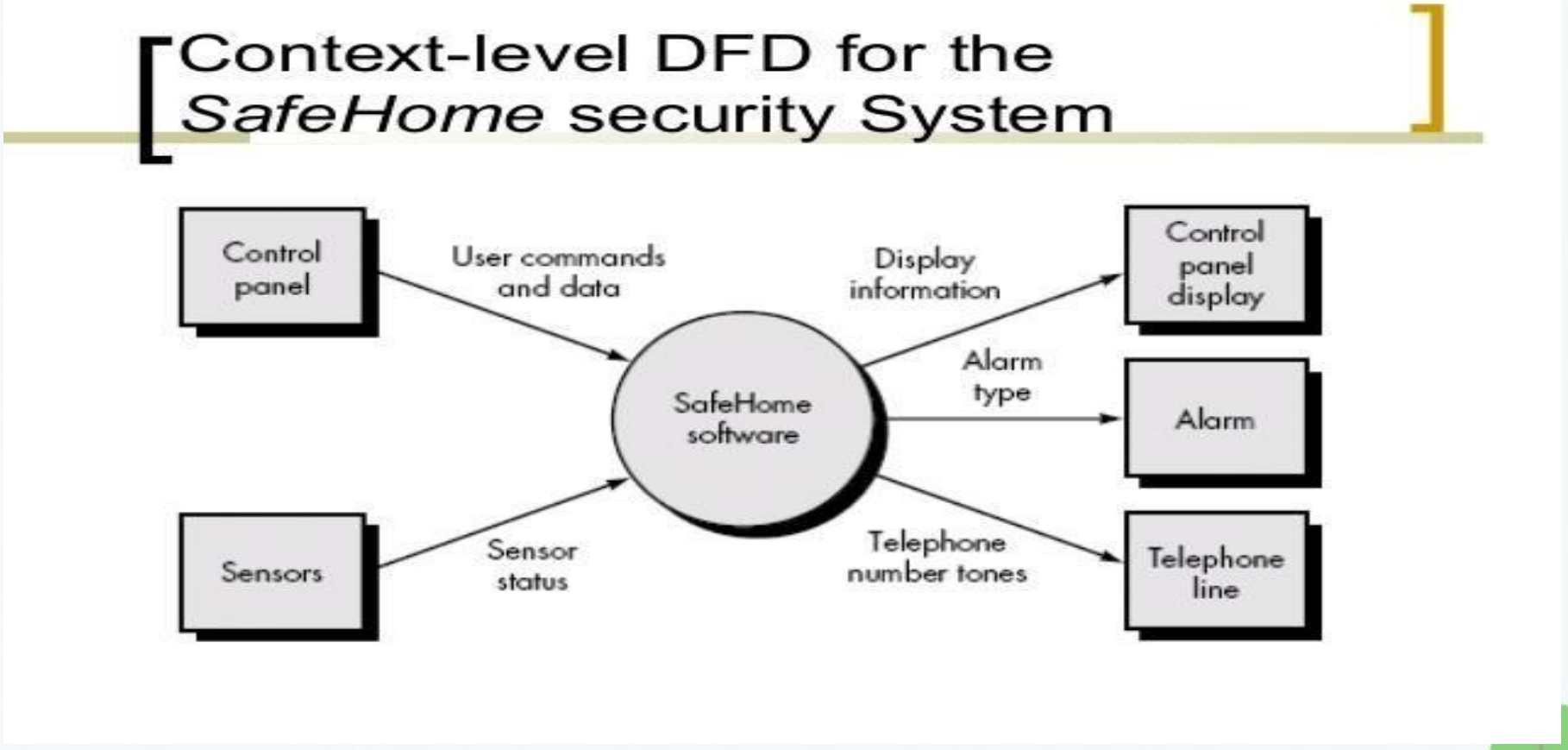
- Software requirements can be mapped into various representations of the design model. The architectural styles represent radically different architectures, so it should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist. In fact, there is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in an ad hoc fashion.
- 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.



- 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; and
- (6) the architectural description is refined and elaborated.

The type of information flow is the driver for the mapping approach required in step 3. In the following sections we examine two flow types.

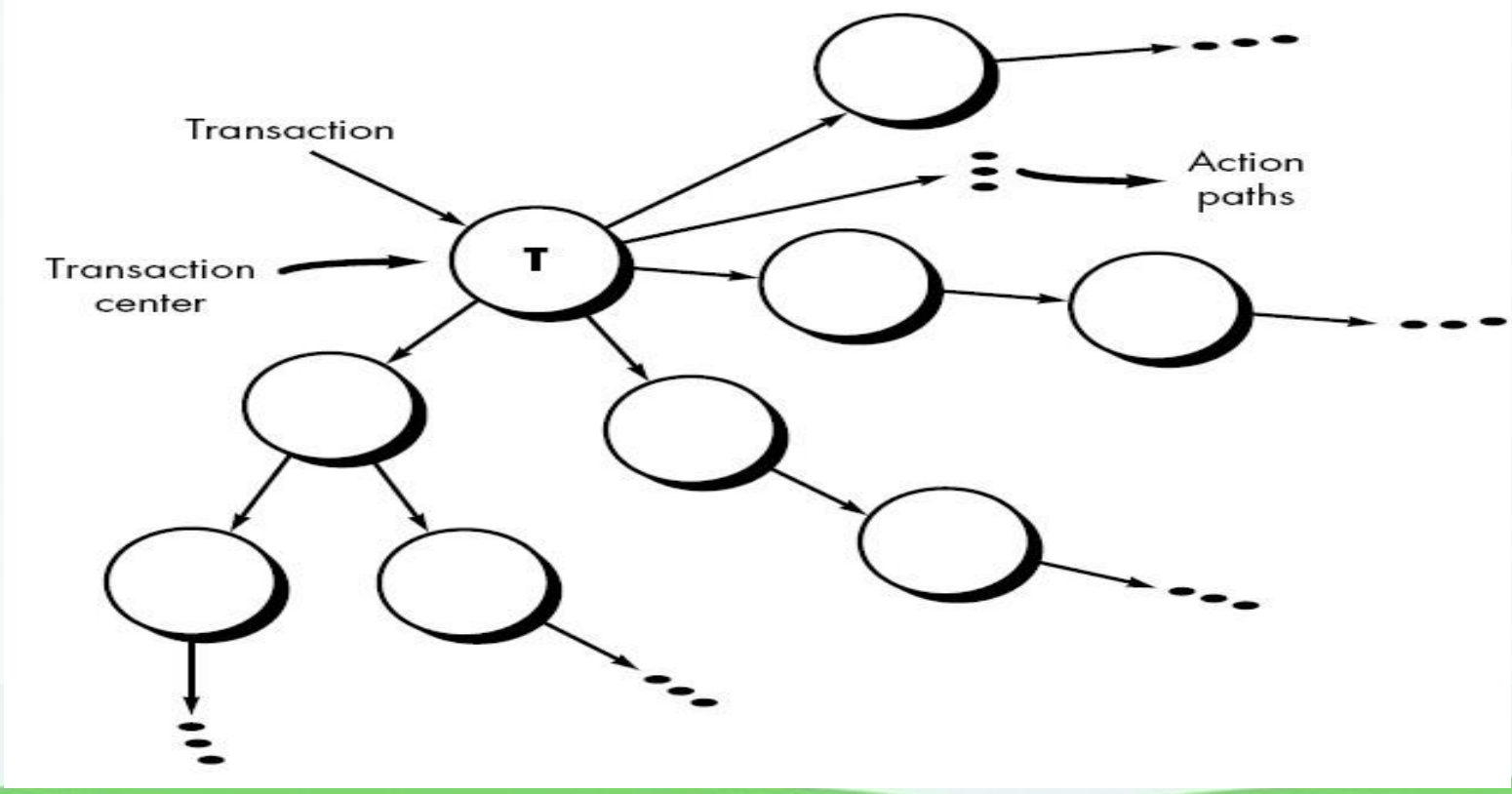
- **Transform Flow**



- Recalling the fundamental system model (level 0 data flow diagram), information must enter and exit software in an "external world" form. For example, data typed on a keyboard, tones on a telephone line, and video images in a multimedia application are all forms of external world information.
- Such externalized data must be converted into an internal form for processing. Information enters the system along paths that transform external data into an internal form. These paths are identified as incoming flow. At the kernel of the software, a transition occurs.
- Incoming data are passed through a transform center and begin to move along paths that now lead "out" of the software. Data moving along these paths are called outgoing flow.
- The overall flow of data occurs in a sequential manner and follows one, or only a few, "straight line" paths. When a segment of a data flow diagram exhibits these characteristics, transform flow is present.

Transaction Flow

- The fundamental system model implies transform flow; therefore, it is possible to characterize all data flow in this category. However, information flow is often characterized by a single data item, called a transaction, that triggers other data flow along one of many paths. When a DFD takes the form shown in figure, transaction flow is present.



- Transaction flow is characterized by data moving along an incoming path that converts xternal world information into a transaction. The transaction is evaluated and, based on its value, flow along one of many action paths is initiated. The hub of information flow from which many action paths emanate is called a transaction center.
- It should be noted that, within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

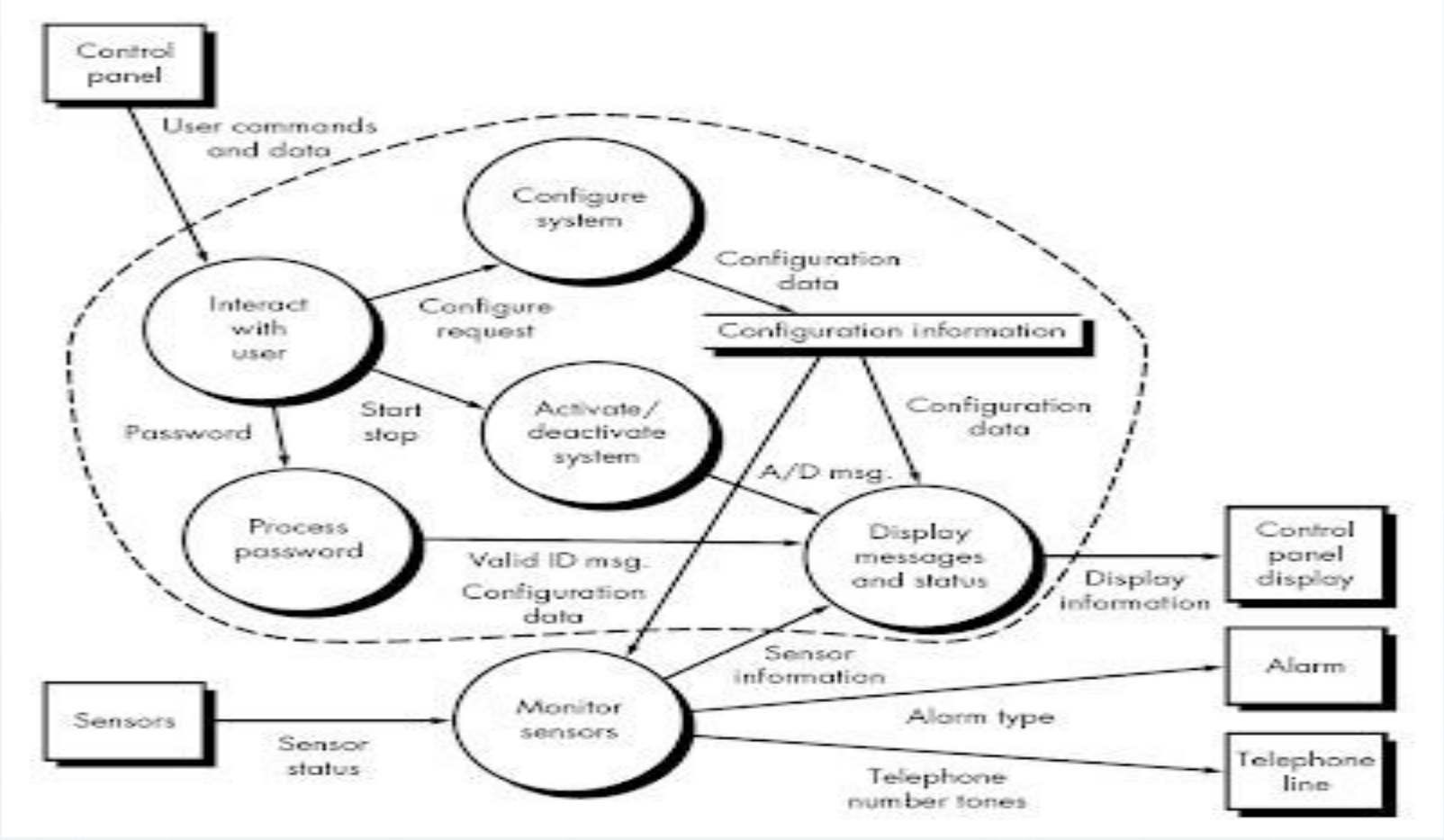
Transform Mapping

- Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. In this section transform mapping is described by applying design steps to an example system—a portion of the SafeHome security software.
- During requirements analysis, more detailed flow models would be created for SafeHome. In addition, control and process specifications, a data dictionary, and various behavioral models would also be created.

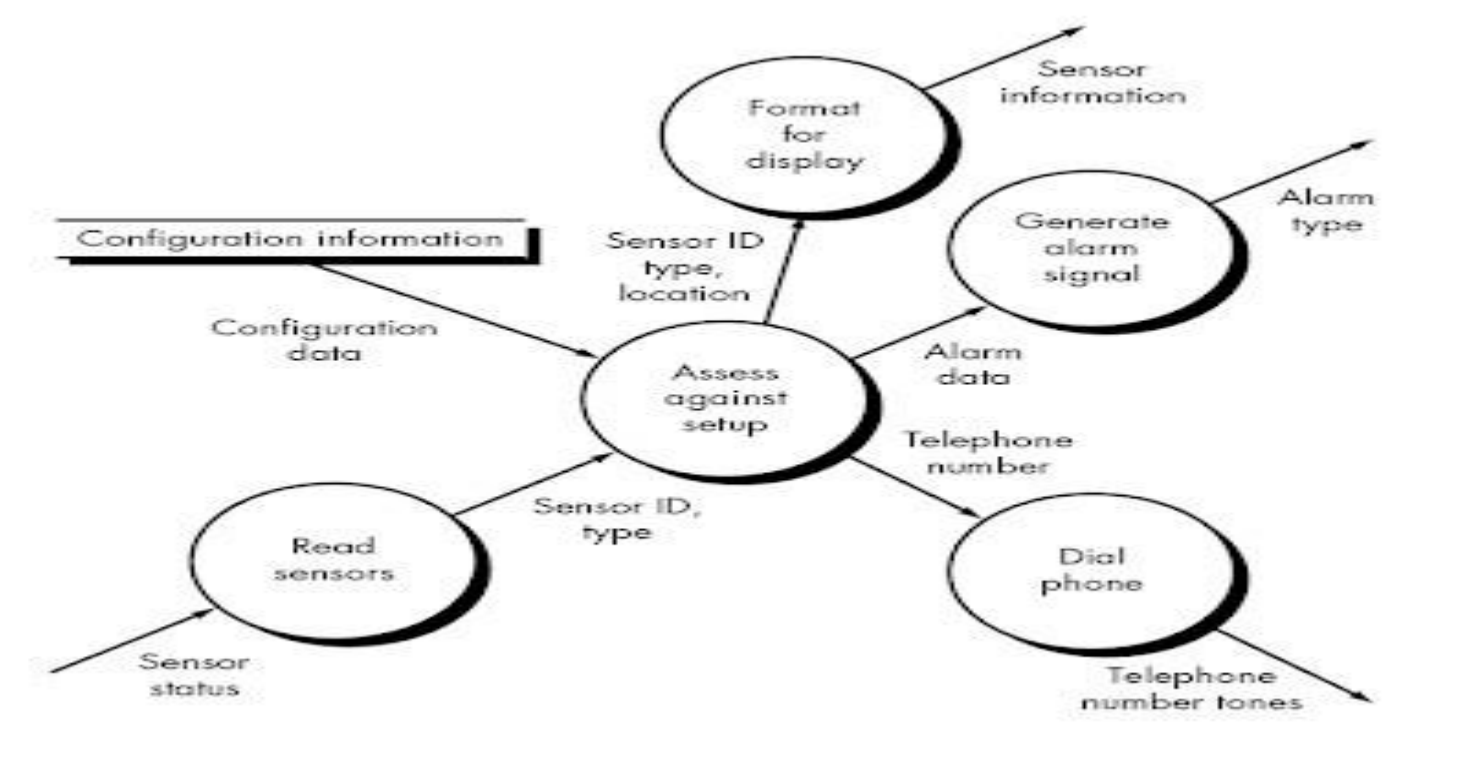
DESIGN STEPS:

- The preceding example will be used to illustrate each step in transform mapping. The steps begin with a re-evaluation of work done during requirements analysis and then move to the design of the software architecture.

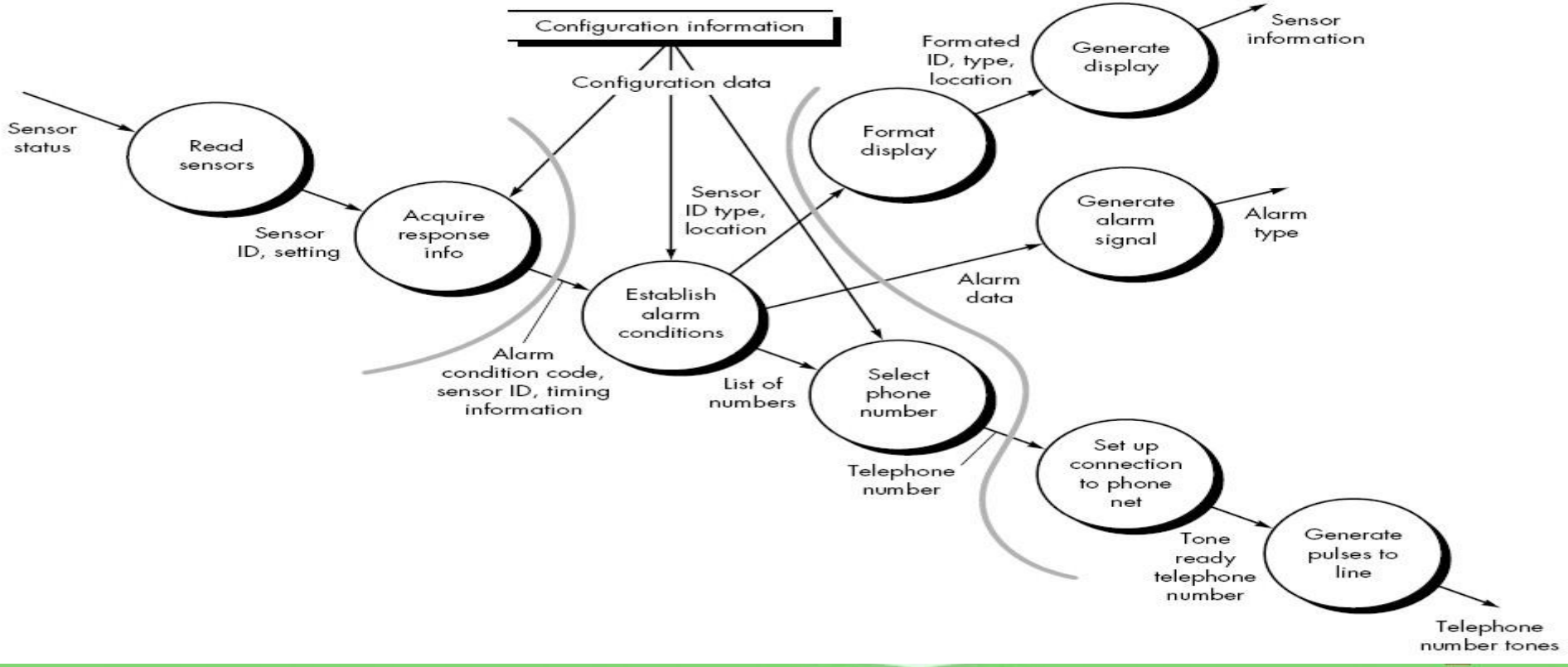
- **Step 1. Review the fundamental system model.** The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the System Specification and the Software Requirements Specification. Both documents describe information flow and structure at the software interface. Figure represent level 1 data flow for the SafeHome software.



- **Step 2. Review and refine data flow diagrams for the software**.
- Information obtained from analysis models contained in the Software Requirements Specification is 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 . At level 3, each transform in the data flow diagram exhibits relatively high cohesion. That is, the process implied by a transform performs a single, distinct function that can be implemented as a module in the SafeHome software. Therefore, the DFD in figure contains sufficient detail for a "first cut" at the design of architecture for the monitor sensors subsystem, and we proceed without further refinement.



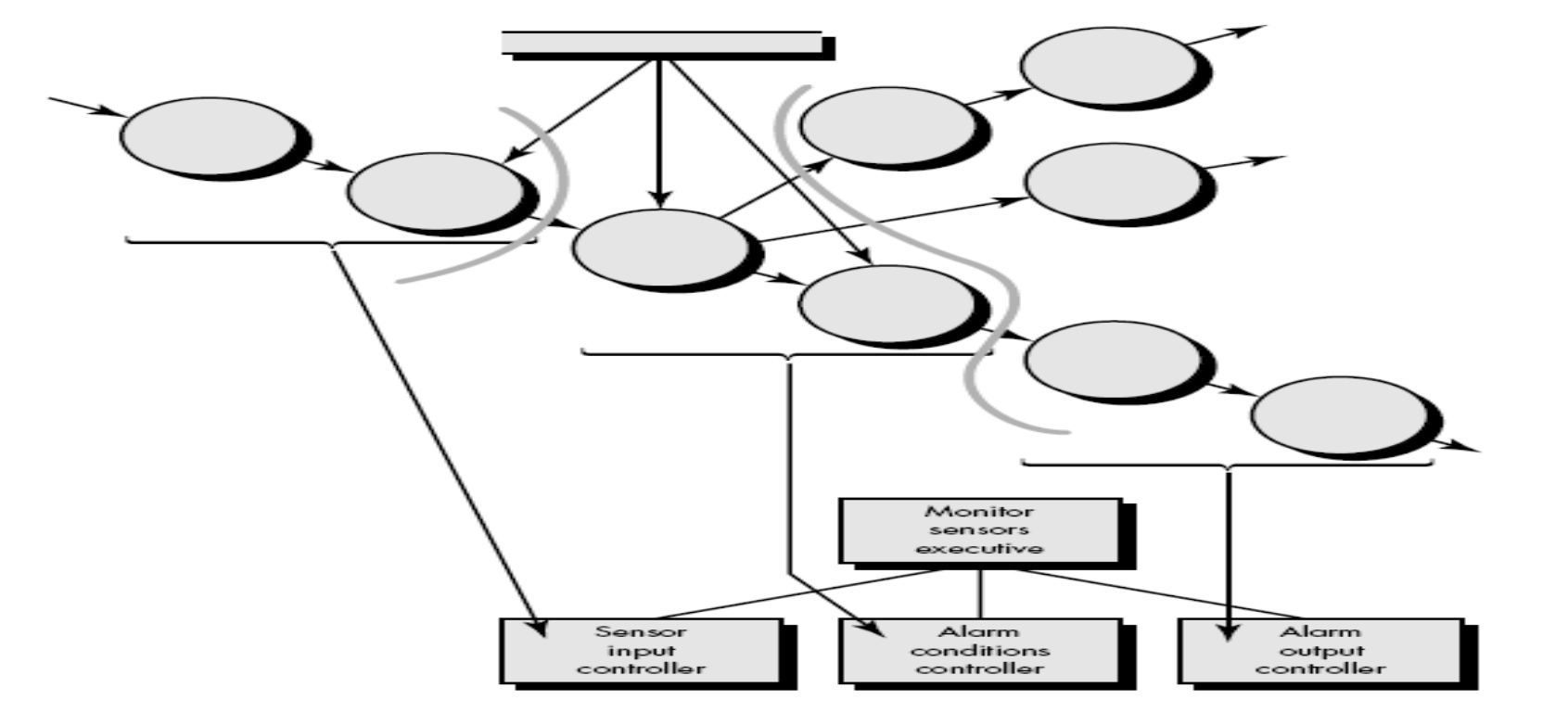
- **Step 3. Determine whether the DFD has transform or transaction flow characteristics.** In general, information flow within a system can always be represented as transform. However, when an obvious transaction characteristic is encountered, a different design mapping is recommended. In this step, the designer selects global (softwarewide) flow characteristics based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These subflows can be used to refine program architecture derived from a global characteristic described previously. For now, we focus our attention only on the monitor sensors subsystem data flow depicted in figure.



- Evaluating the DFD , we see data entering the software along one incoming path and exiting along three outgoing paths. No distinct transaction center is implied (although the transform establishes alarm conditions that could be perceived as such). Therefore, an overall transform characteristic will be assumed for information flow.

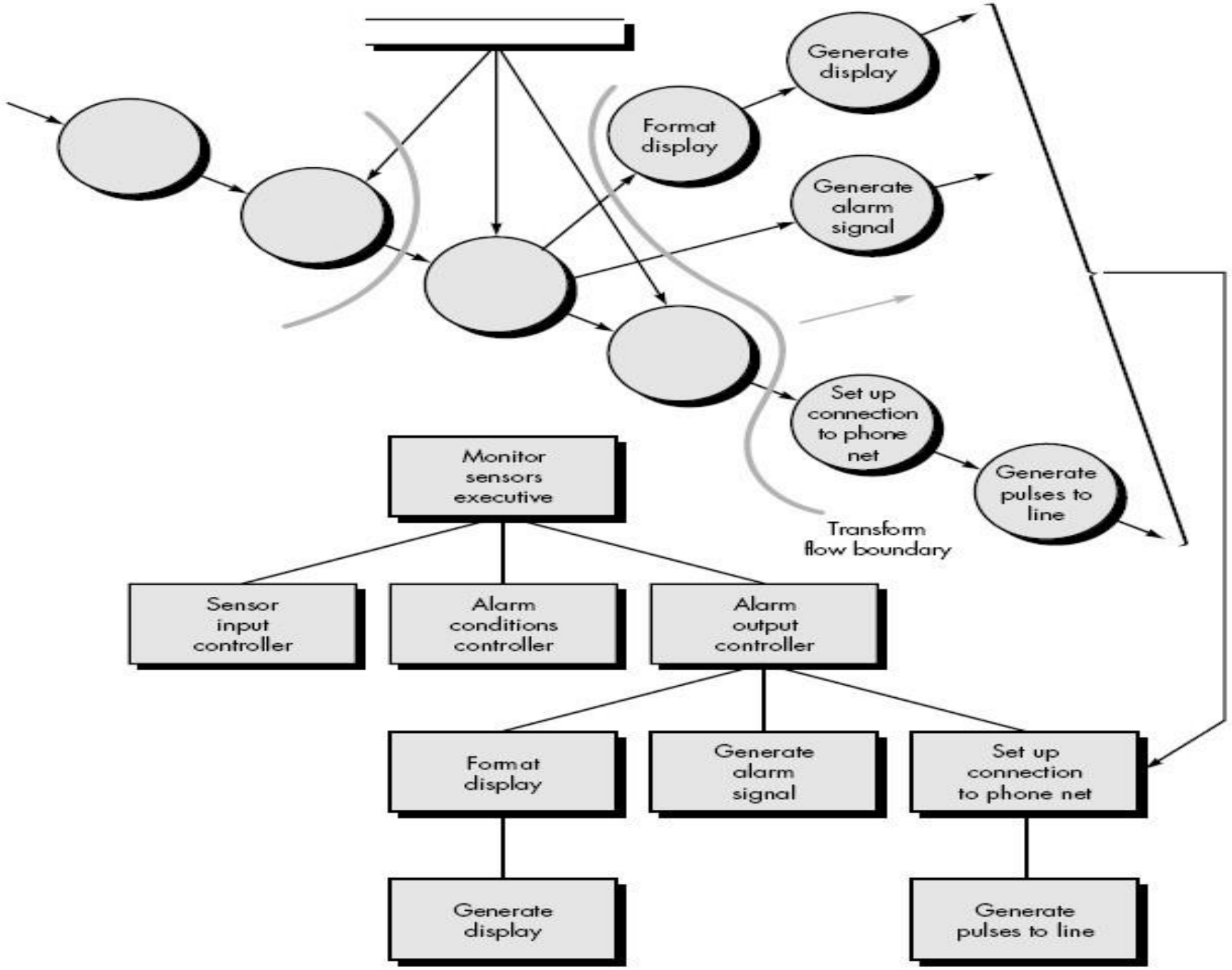
- **Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries.**
- In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.
- Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in the above figure. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary (e.g, an incoming flow boundary separating read sensors and acquire response info could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

- **Step 5. Perform "first-level factoring." Program structure represents a top-down distribution of control**
- Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work.
- 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. This first-level factoring for the monitor sensors subsystem is illustrated in figure below. A main controller (called monitor sensors executive) resides at the top of the program structure and coordinates the following subordinate control functions:
- **• An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.**
- **• 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.**



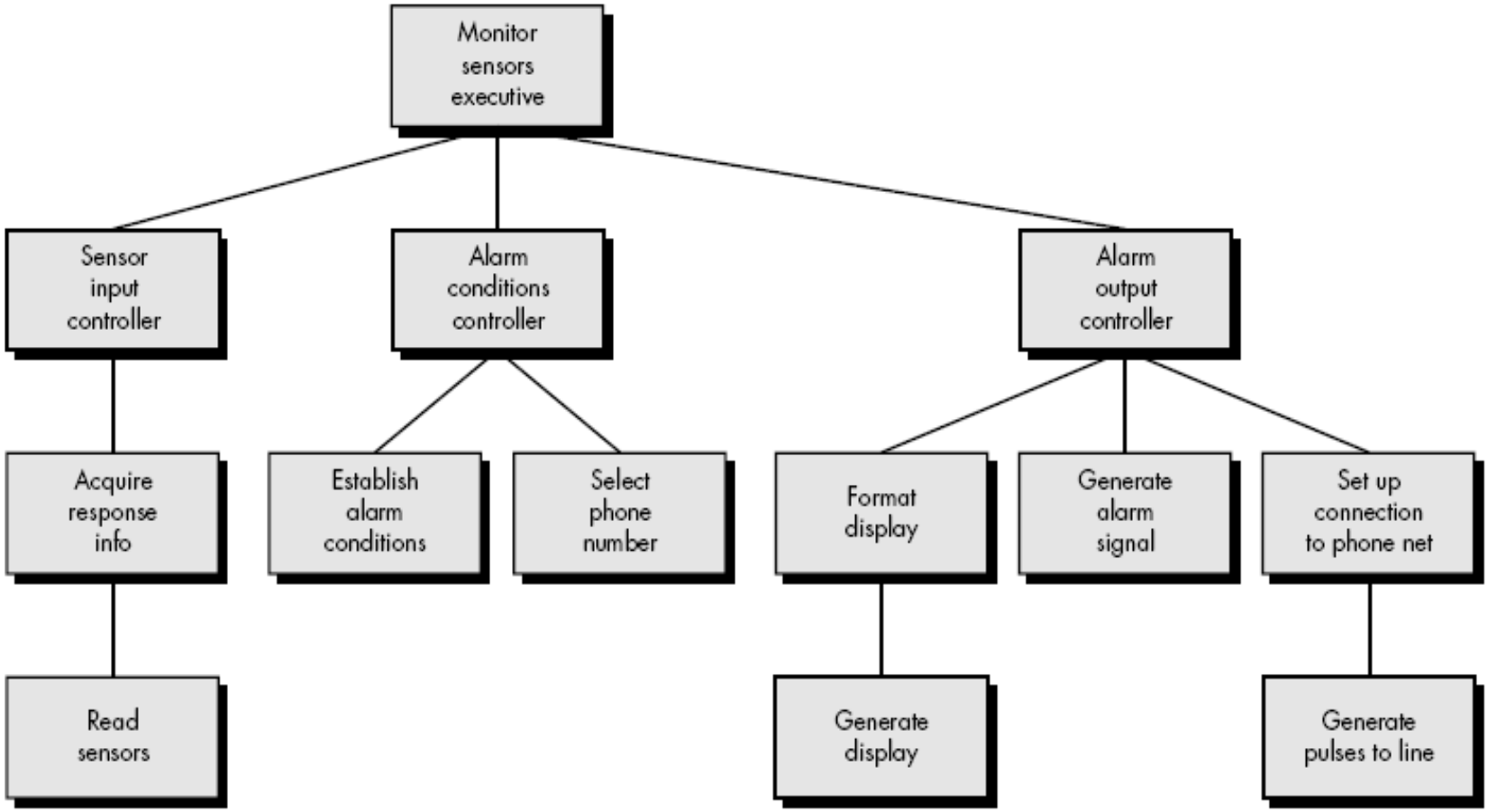
- Although a three-pronged structure is implied by figure complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good coupling and cohesion characteristics.

- **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.** Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second-level factoring for the SafeHome data flow is illustrated in figure.


Generate display
Format display
Generate alarm signal
Set up connection to phone net
Generate pulses to line
Transform flow boundary
Monitor sensors executive
Sensor input controller
Alarm conditions controller
Alarm output controller
Format display
Generate alarm signal
Set up connection to phone net
Generate display
Generate pulses to line

- Although the figure 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. Practical considerations and measures of design quality dictate the outcome of secondlevel factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration" design.

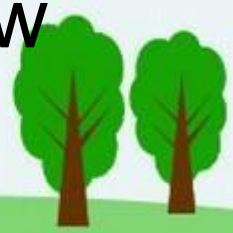
- Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of monitor sensors subsystem software is mapped somewhat differently. Each of the data conversion 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.


Monitor sensors executive
Sensor input controller
Alarm conditions controller
Alarm output controller
Acquire response info
Establish alarm conditions
Select phone number
Format display
Generate alarm signal
Set up connection to phone net
Read sensors
Generate display
Generate pulses to line

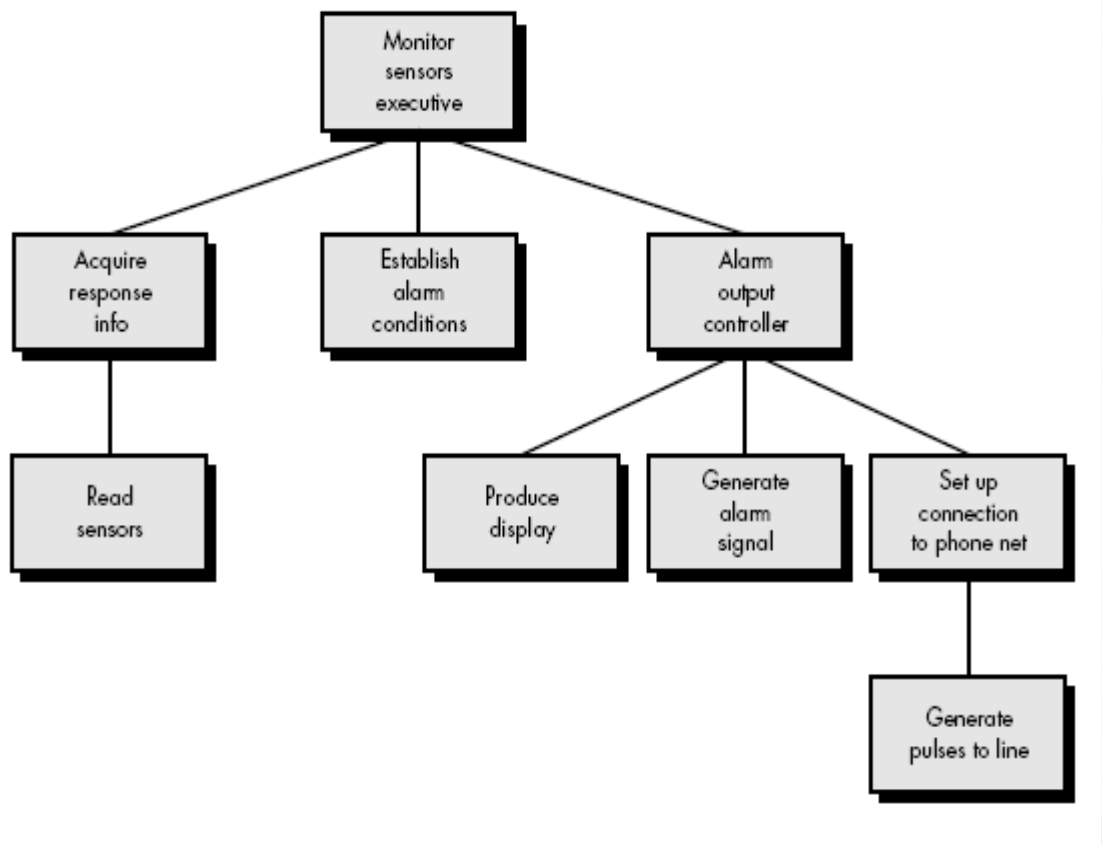
- The modules mapped in the preceding manner and shown in figure represent an initial design of software architecture. Although modules are named in a manner that implies function, a brief processing narrative (adapted from the PSPEC created during analysis modeling) should be written for each. The narrative describes
- Information that passes into and out of the module (an interface description).
- Information that is retained by a module, such as data stored in a local data structure.
- A procedural narrative that indicates major decision points and tasks.
- A brief discussion of restrictions and special features (e.g., file I/O, hardware dependent characteristics, special timing requirements).
- The narrative serves as a first-generation Design Specification. However, further refinement and additions occur regularly during this period of design.

- **Step 7. Refine the first-iteration architecture using design heuristics for improved software quality.** A first-iteration architecture can always be refined by applying concepts of module independence . Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.
- Refinements are dictated by the analysis and assessment methods described briefly , as well as practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a module that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved.
- Software requirements coupled with human judgment is the final arbiter. Many modifications can be made to the first iteration architecture developed for the SafeHome 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.



- The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

No, writing a program is the different concept in design software.

Design is the place where software quality is established. Before starting of design software, first requirements should be analyzed and specified. In the software design process, design engineering is the one of the concept. While beginning software, requirements have been analyzed and modeled. This model can be accessed for quality and improved before code is generated. In a software engineering context, first need to develop the models of program. Not the program themselves.

Software design different from coding: At first it is very clear that, design is not coding and coding is not design. It is created from program components. Design is the description of the logic, which is used in solving the problem. Coding is the language specification which is implementation of the design. It runs on the computer and, provides the expected result.