Chapter 9 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. The data, functional, and behavioral domains serve as a guide for the creation of the software design.

a) **What is it?**

Architectural design represents the structure of data and program components that are required to build a computer-based system.

b) **Who does it?**

Specialists-- when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements.

c) **Why is it important?**

It provides you with the big picture and ensures that you've got it right.

d) **What are the steps?**

It begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative styles are also analyzed to derive the structure that is best suited to customer requirements and quality attributes. After that, the architecture is elaborated using an architectural design method.

e) **What is the work product?**

An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

f) **How do I ensure that I've done it right?**

At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

9.1 SOFTWARE ARCHITECTURE

9.1.1 What Is Architecture?

In terms of building, it is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity.

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

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

(1) Analyze the effectiveness of the design to meet requirements,

(2) Consider architectural alternatives to handle design changes easily, and

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

In the context of architectural design, a software component can be a program module or an object-oriented class, but it can be extended to include databases and "middleware" that enable the configuration of a network of clients and servers.

The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. **At the architectural level, internal properties (e.g., details of an algorithm) are not specified.** The relationships between components can be as simple as a function call from one module to another or as complex as a database access protocol.

- **Difference between the terms architecture and design**

  A design is an instance of an architecture For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform or Microsoft platform. So, there is one architecture, but many designs can be created based on that architecture.

Design of software architecture considers two levels of the design pyramid—data design and architectural design.

Data design enables you to represent the data component of the architecture in conventional systems.

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

9.6 ARCHITECTURAL MAPPING USING DATA FLOW

A comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist.

A mapping technique, called **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. It 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 the program structure,

(4) Control hierarchy is defined,

(5) The resultant structure is refined using design measures and heuristics, and

(6) The architectural description is refined and elaborated.

In order to perform the mapping, the type of information flow must be determined. One type of information flow is called transform flow. Data flows into the system along an incoming flow path. Then it is processed at a transform center. Finally, it flows out of the system along an outgoing flow path that transforms the data into external world form.

9.6.1 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. To map these data flow diagrams into a software architecture, you would initiate the following design steps: (Example Home security System)

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 9.10 depicts a level 0 context model, and Figure 9.11 shows refined data flow for the security function.

FIGURE 9.10

Context-level DFD for the *SafeHome* security function

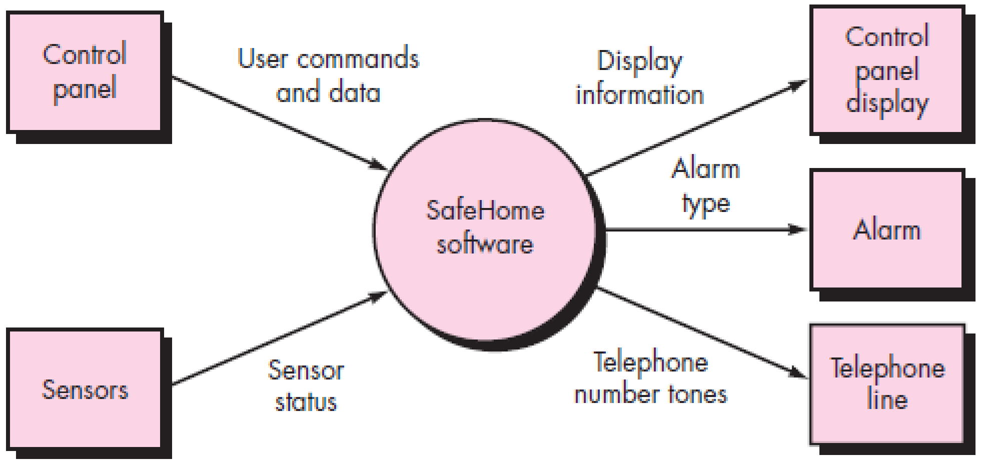
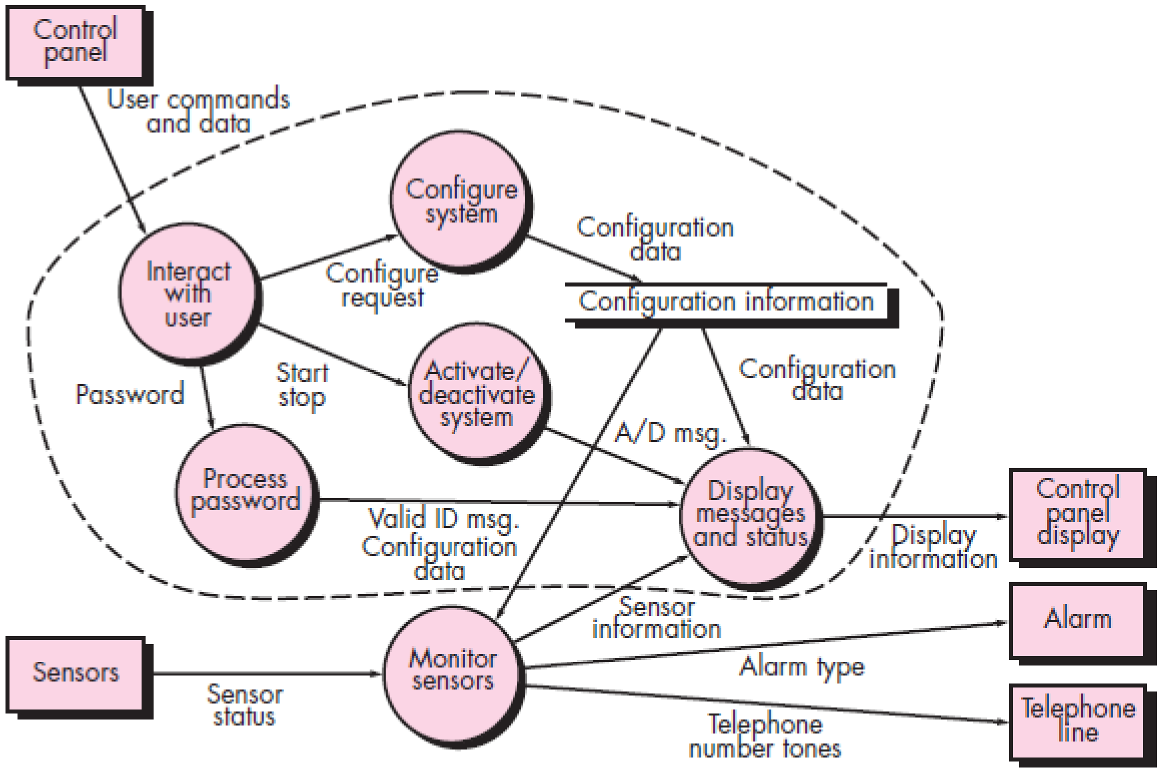


FIGURE 9.11

Level 1 DFD for the *SafeHome* security function



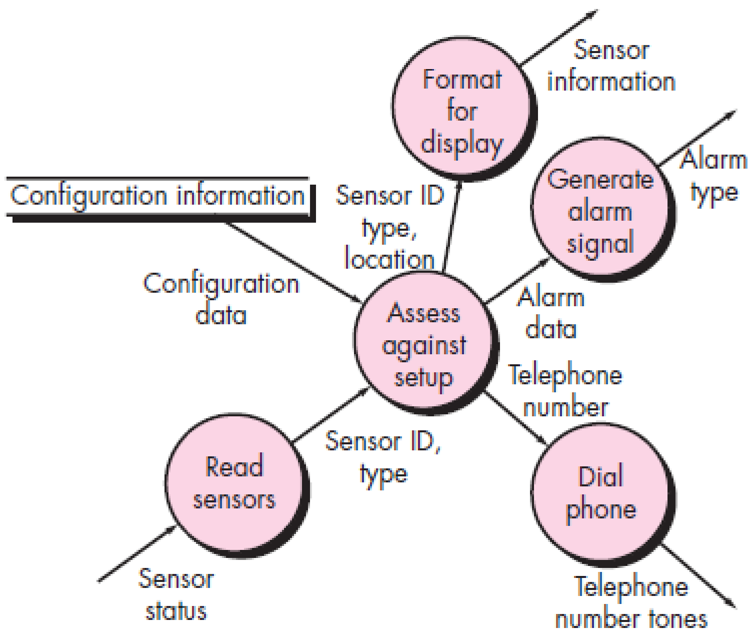
Step 2. Review and refine data flow diagrams for the software.

Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors

Reference:- Roger S Pressman - Software engineering _ a practitioner's approach-McGraw-Hill Higher Education (2010)

FIGURE 9.12

Level 2 DFD that refines the *monitor sensors* transform



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

Evaluating the DFD. Input and output should be consistent for a process.

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

Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. 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. 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."

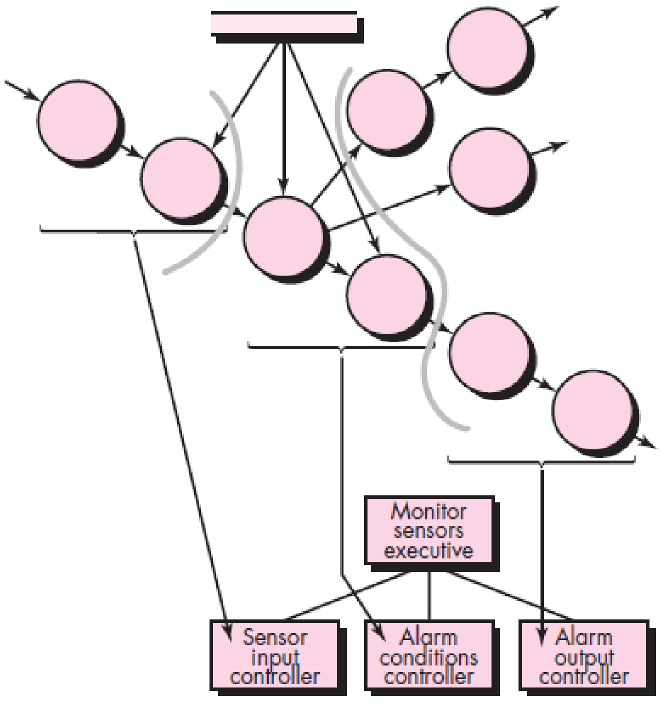
This mapping is top-down distribution of control. Factoring leads to a program structure in which

- ✓ Top-level components perform decision making and
- ✓ Low-level components perform most input, computation, and output work.
- ✓ Middle-level components 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 9.14.

FIGURE 9.14

First-level factoring for *monitor sensors*



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.

Step 6. Perform "second-level factoring."

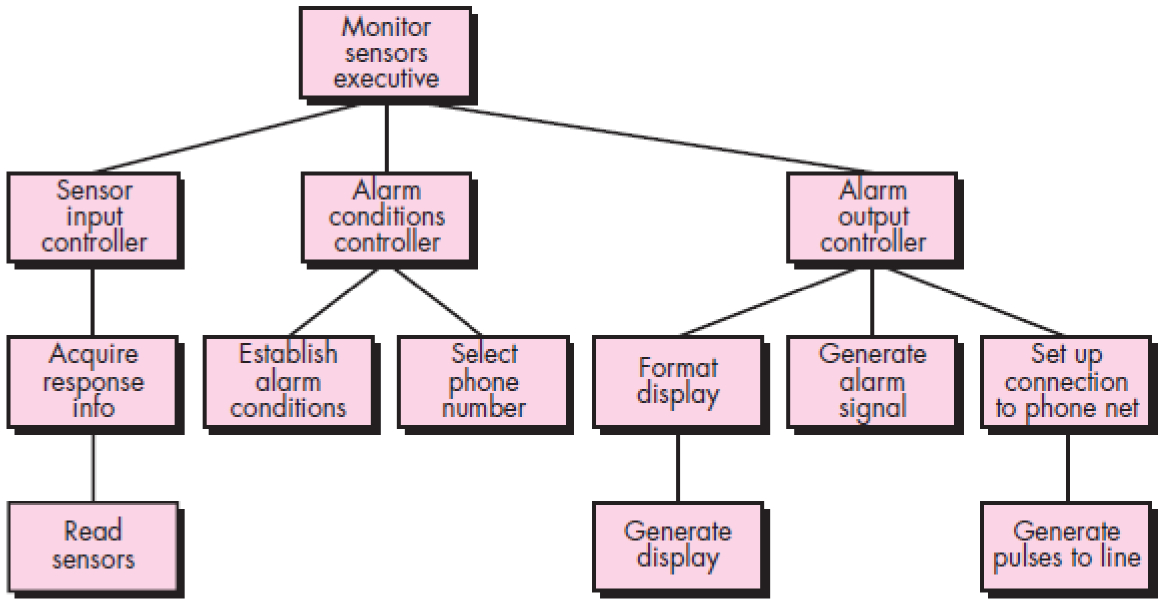
Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure.

Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components. 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. A completed first-iteration architecture is shown in Figure 9.16.

FIGURE 9.16

First-iteration structure for *monitor sensors*



Components are named in a manner that implies function. The processing narrative describes the component interface, internal data structures, a functional narrative, and a brief discussion of restrictions and special features.

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 functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

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.