MODULE II

Design process and concepts – modular design – design heuristic –design model and document, Architectural design – software architecture – data design –architectural design - transform and transaction mapping – user interface design – user interface design principles. Real time systems – Real time software design – system design – real time executives – data acquisition system – monitoring and control system. SCM – Need for SCM – Version control – Introduction to SCM process – Software configuration items

Topic	Sub Topic	Table Of Contents
1		Design Process
2		Design Concepts
3		Modular Design
4		Design Heuristic
	4.1	Design Heuristic For Effective Modularity
	4.2	Design Model
	4.3	Design Document
5		Architectural Design
	5.1	Software Architecture
	5.2	Defining Software Architectural Design
	5.3	Data Design
	5.4	Architectural mapping using Data Flow - Transform And Transaction Mapping
6		User Interface Design
	6.1	UID Principles
7		Real Time Systems
	7.1	Real Time Software Design
	7.2	System Design
	7.3	Real Time Executives
	7.4	Monitoring And Control Systems
	7.5	Data Acquisition Systems
8		Software Change Management SCM
	8.1	Need For SCM
	8.2	Introduction To SCM Process
9		Version Control
10		Software Configuration Items

1. DESIGN PRINCIPLES

- **The design process should not suffer from "tunnel vision."** A good designer should consider alternative approaches, judging each based on the requirements of the problem, and the resources available to do the job.

- **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- **The design should not reinvent the wheel.** Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
- **The design should "minimize the intellectual distance"** between the software and the problem as it exists in the real world. That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- **The design should exhibit uniformity and integration.** A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
- **The design should be structured to accommodate change.** The design concepts discussed in the next section enable a design to achieve this principle.
- **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Well designed software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
- **Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
- **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts and design measures are available to assist the designer in assessing quality.
- **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

2. DESIGN CONCEPTS

2.1 Abstraction

"Abstraction is one of the fundamental ways to cope with complexity." Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a

computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

There are different levels of Abstraction, *Procedural Abstraction and Data Abstraction*.

- 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. In the context of the procedural abstraction open, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, 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**.

2.2 Refinement

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

Refinement is actually a process of elaboration. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

2.3 Modularity

Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called modules that are integrated to satisfy problem requirements. *"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 reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

If we subdivide software indefinitely, the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure below, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grow. These characteristics lead to a total cost or effort

curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

Note: Don't over modularize. The simplicity of each module will be over shadowed by the complexity of integration.

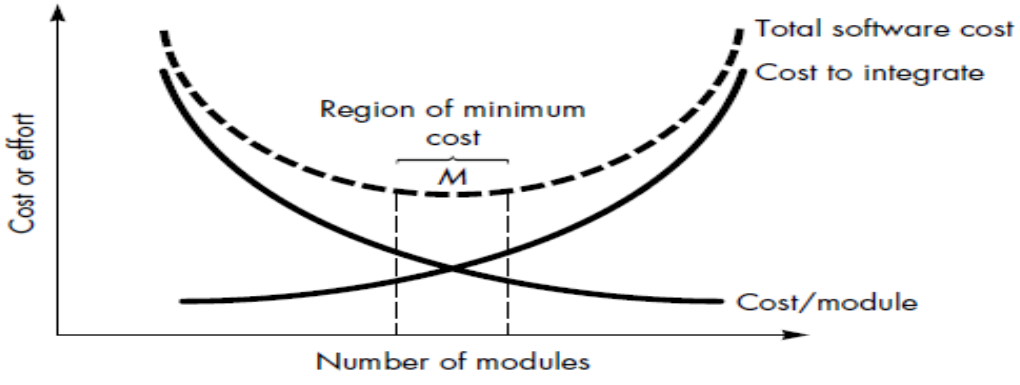


Figure: Modularity and software cost

Important question arises when modularity is considered. How can we evaluate a design method to determine if it will lead to effective modularity?

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.

2.4 Software Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

Note: "Software architecture is the development work product that gives the highest return on investment with respect to quality, schedule and cost."

Five different types of models are used to represent the architectural design.

a. *Structural models* represent architecture as an organized collection of program components.
b. *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.
c. *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
d. *Process models* focus on the design of the business, or technical process that the system must accommodate.
e. *Functional models* can be used to represent the functional hierarchy of a system.

2.5 Control Hierarchy

Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control. Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation. The most common is the tree like diagram, figure below that represents hierarchical control for call and return architectures.

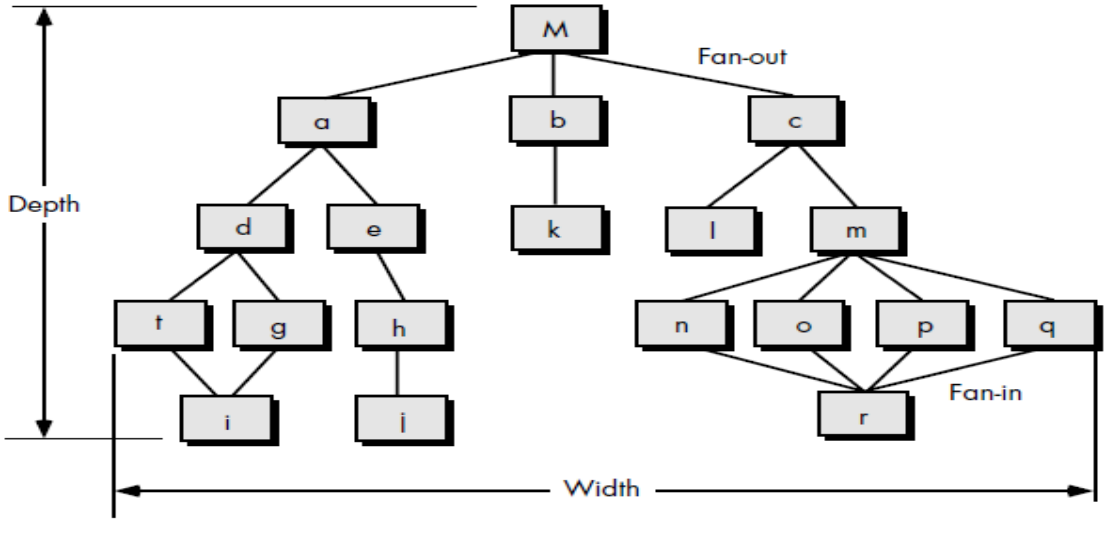


FIGURE: Structure terminology for a call and return architectural style

Referring to Figure, depth and width provide an indication of the number of levels of control and overall span of control, respectively. Fan-out is a measure of the number of modules that are directly controlled by another module. Fan-in indicates how many modules directly control a given module.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be ***superordinate*** to it, and conversely, a module controlled by another is said to be ***subordinate*** to the controller. For example, referring to Figure, module M is superordinate to modules a, b, and c. Module h is subordinate to module e and is ultimately subordinate to module M. *Width-oriented*

relationships (e.g., between modules d and e) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: **visibility and connectivity**. *Visibility* indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. *Connectivity* indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.

2.6 Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to *Figure a*, horizontal partitioning defines separate branches of the modular hierarchy for each major program function. Control modules, represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called processing) and output.

What are the benefits of horizontal partitioning?

Partitioning the architecture horizontally provides a number of distinct benefits:

- Software that is easier to test
- Software that is easier to maintain
- Propagation of fewer side effects
- Software that is easier to extend

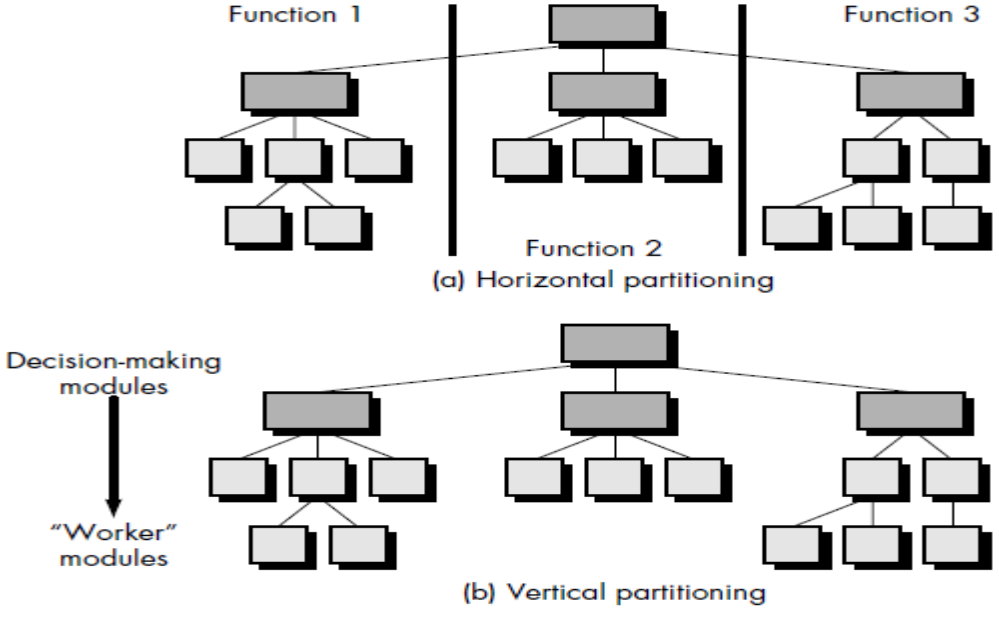


Figure: Structural Partitioning

Vertical partitioning Figure b, often called factoring, suggests that control (decision making) and work should be distributed top-down in the program structure. Top-level modules should perform control functions and do little actual processing work.

Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

Note: "Worker" modules tend to change more frequently than control modules. By placing the workers low in the structure, side effects (due to change) are reduced.

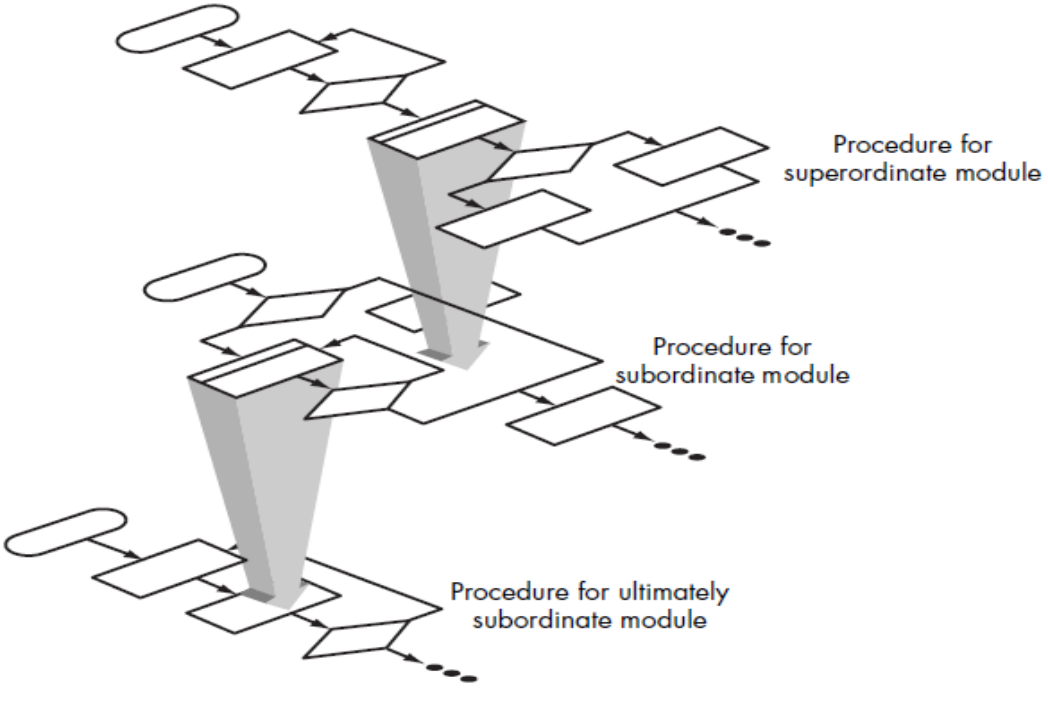
2.7 Data Structure

- Data structure is a representation of the logical relationship among individual elements of data.
- Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information.

2.8 Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

Figure: Procedure is layered



2.9 Information Hiding

The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, 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 use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. i.e propagation of errors are less likely to happen, since information is hidden.

3. MODULAR DESIGN

A modular design reduces complexity, facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

3.1 Functional Independence

The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding. Need to design software so that each module addresses a specific sub function of requirements and has a simple interface when viewed from other parts of the program structure.

Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

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

3.1.1 Cohesion

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is logically cohesive. When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits temporal cohesion.

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed pre-specified bounds. It performs the following tasks:

(1) Computes supplementary data based on original computed data,
(2) Produces an error report (with graphical content) on the user's workstation,
(3) Performs follow-up calculations requested by the user,
(4) Updates a database, and
(5) Enables menu selection for subsequent processing.

High cohesion is characterized by a module that performs one distinct procedural task.

3.1.2 Coupling

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

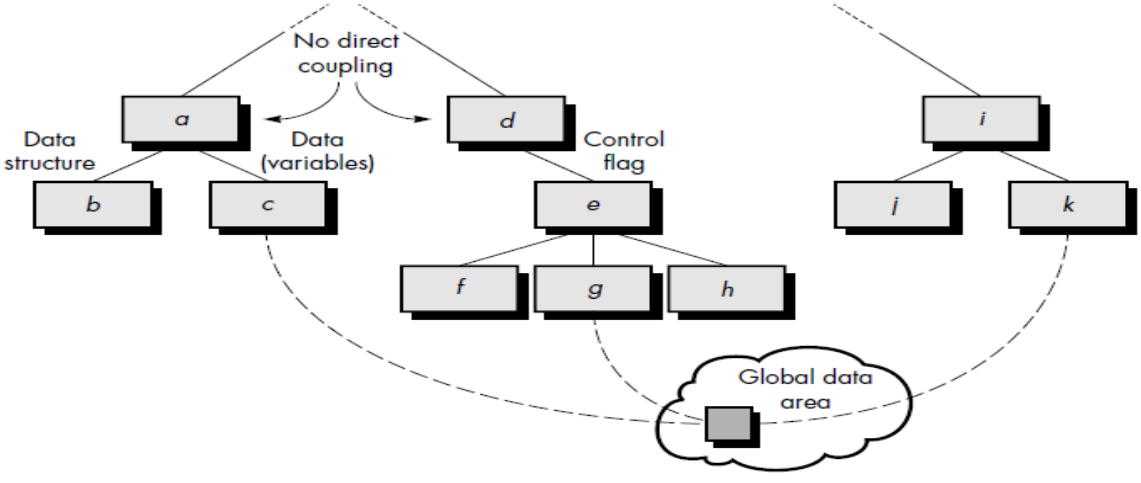


Figure: Types of Coupling

Figure above provides examples of different types of module coupling. Modules a and d are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module c is subordinate to module a and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present, *low coupling* (called data coupling) is exhibited in this portion of structure. A variation of data coupling, called *stamp coupling* is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules b and a.

At moderate levels, coupling is characterized by passage of control between modules. *Control coupling* is very common in most software designs and is shown in the figure above where a "control flag" (a variable that controls decisions in a subordinate or superordinate module) is passed between modules d and e.

Relatively high levels of coupling occur when modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols. *External coupling* is essential, but should be limited to a small number of modules with a structure. High coupling also occurs when a number of modules reference a global data area.

Common coupling, as this mode is called, is shown in the figure. Modules c, g, and k each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module c initializes the item. Later module g recomputes and updates the item. Let's assume that an error occurs and g updates the item incorrectly. Much later in processing module, k reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module k; the actual cause, module g.

Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them.

The highest degree of coupling, content coupling, occurs when one module makes use of data or control information maintained within the boundary of another module. Secondarily, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

4. DESIGN HEURISTICS

4.1 DESIGN HEURISTICS FOR EFFECTIVE MODULARITY

Once program structure has been developed, effective modularity can be achieved by applying the design concepts introduced earlier in this chapter. 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. Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence. An exploded module becomes two or more modules in the final program structure. An imploded module is the result of combining the processing implied by two or more modules. An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. 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. The structure shown inside the cloud in the figure below does not make effective use of factoring. All modules are “pancaked” below a single control module. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

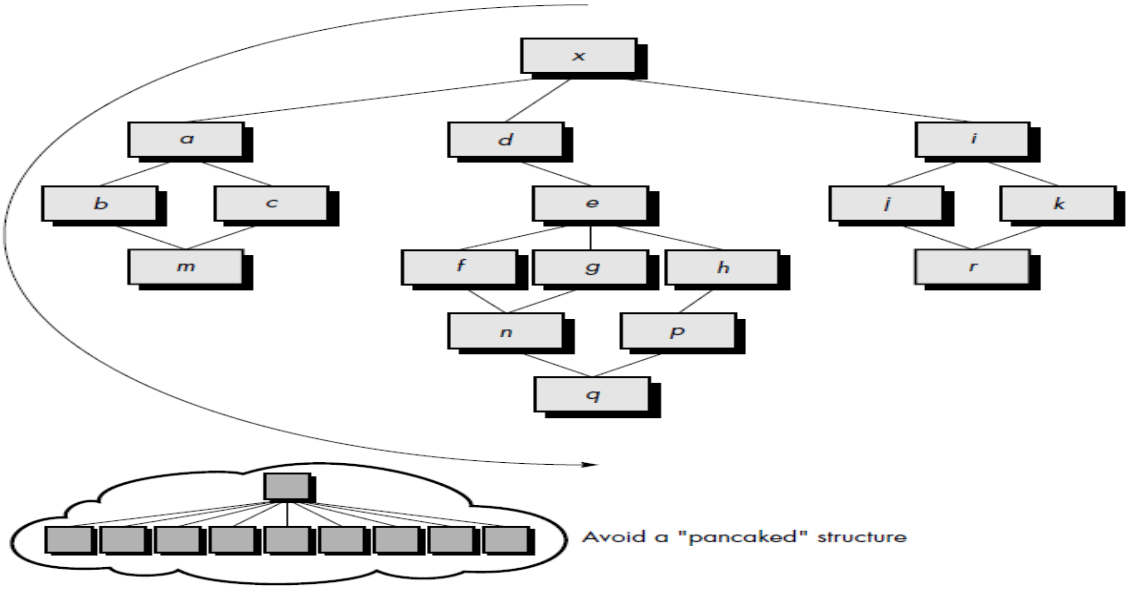


Figure: Program Structures

3. Keep the scope of effect of a module within the scope of control of that module. The scope of effect of module e is defined as all other modules that are affected by a decision made in module e. The scope of control of module e is all modules that are subordinate and ultimately subordinate to module e. Referring to Figure: Program Structures, 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.

4. Evaluate module interfaces to reduce complexity and redundancy and improve consistency. Module interface complexity is a prime cause of software errors. Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be revaluated.

5. Define modules whose function is predictable, but avoid modules that are overly restrictive. 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. Modules that have internal "memory" can be unpredictable unless care is taken in their use.

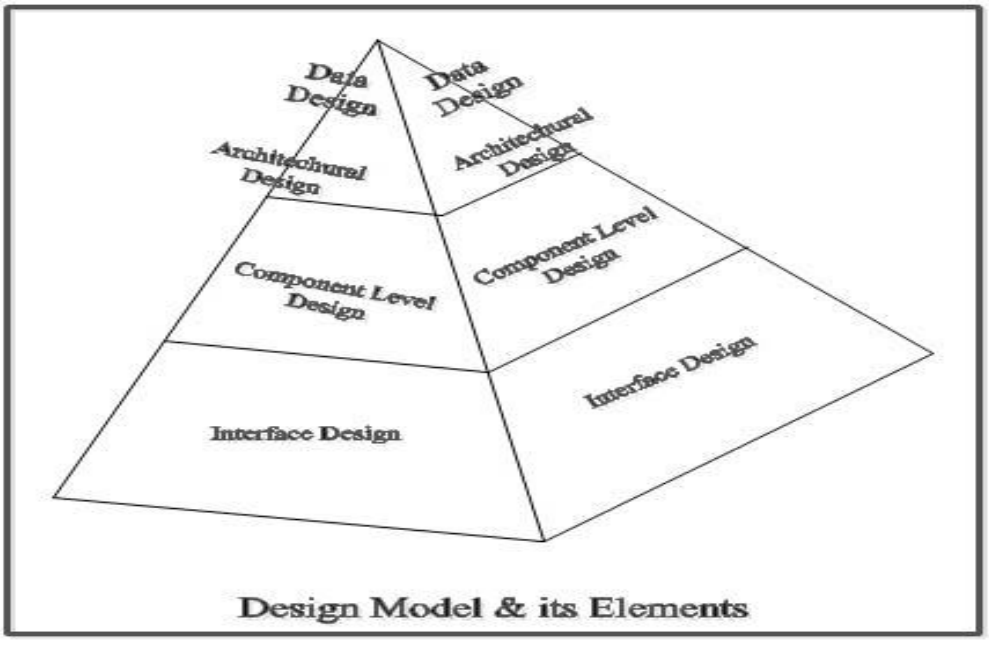
A module that restricts processing to a single sub function exhibits high cohesion and is viewed with favour by a designer. However, a module that arbitrarily restricts the size of a local data structure, options within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

6. Strive for "controlled entry" modules by avoiding "pathological connections." This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. Pathological connection refers to branches or references into the middle of a module.

4.2 Developing a Design Model

To develop a complete specification of design (design model), four design models are needed. These models are listed below.

1. **Data design:** This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design models.
2. **Architectural design:** This specifies the relationship between the structural elements of the software, design patterns, architectural styles, and the factors affecting the ways in which architecture can be implemented.
3. **Component-level design:** This provides the detailed description of how structural elements of software will actually be implemented.
4. **Interface design:** This depicts how the software communicates with the system that interoperates with it and with the end-users.



Design Model & its Elements

4.3 Software design Document (description)

- A software design document or SDD is a written description of a software product, that a software designer writes in order to give a software development team overall guidance to the architecture of the software project.
- An SDD usually accompanies an architecture diagram with pointers to detailed feature specifications of smaller pieces of the design. Practically, the description is required to coordinate a large team under a single vision, needs to be a stable reference, and outline all parts of the software and how they will work.

The SDD usually contains the following information:

- The data design describes structures that reside within the software. Attributes and relationships between data objects dictate the choice of data structures.
- The architecture design uses information flowing characteristics, and maps them into the program structure. The transformation mapping method is applied to exhibit distinct boundaries between incoming and outgoing data. The data flow diagrams allocate control input, processing and output along three separate modules.
- The interface design describes internal and external program interfaces, as well as the design of human interface. Internal and external interface designs are based on the information obtained from the analysis model.
- The procedural design describes structured programming concepts using graphical, tabular and textual notations. These design mediums enable the designer to represent procedural detail, that facilitates translation to code.

This blueprint for implementation forms the basis for all subsequent software engineering work.

5. Architectural Design

Ian Sommerville

Why Is Architecture Important?

Three key reasons for software architecture is important are:

1. Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
2. 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.
3. Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together". Establishing the overall structure of a software system

5.1.1 Definitions

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is ***Architectural Design***
- The output of this design process is a description of the ***Software Architecture***

Architectural design

- An early stage of the system design process
- Represents the link between specification and design processes
- Often carried out in parallel with some specification activities
- It involves identifying major system components and their communications

5.1.2 Advantages of explicit Architecture

- Stakeholder Communication
  - Architecture may be used as a focus of discussion by system stakeholders
- System Analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible
- Large-scale Reuse
  - The architecture may be reusable across a range of systems

5.1.3 Architectural design process

- System Organisation
  - The system is decomposed into several principal sub-systems and communications between these sub-systems are identified

- Modular decomposition styles
  - The identified sub-systems are decomposed into modules
- Control styles
  - A model of the control relationships between the different parts of the system is established

5.1.4 Sub-systems and Modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system

5.1.5 Architectural models – Definition

- Different architectural models may be produced during the design process
- Each model presents different perspectives on the architecture

5.1.6 Architectural models include

- ***Static structural model*** that shows the major system components
- ***Dynamic process model*** that shows the process structure of the system
- ***Interface model*** that defines sub-system interfaces
- ***Relationships model*** such as a data-flow model
- ***Distribution model*** that shows sub-system distribution

5.1.7 Architectural styles

- The architectural model of a system may conform to a generic architectural model or style
- An awareness of these styles can simplify the problem of defining system architectures
- However, most large systems are heterogeneous and do not follow a single architectural style

5.1.8 Architecture attributes

- Performance
  - Localise operations to minimise sub-system communication
- Security
  - Use a layered architecture with critical assets in inner layers
- Safety
  - Isolate safety-critical components
- Availability
  - Include redundant components in the architecture
- Maintainability
  - Use fine-grain, self-contained components

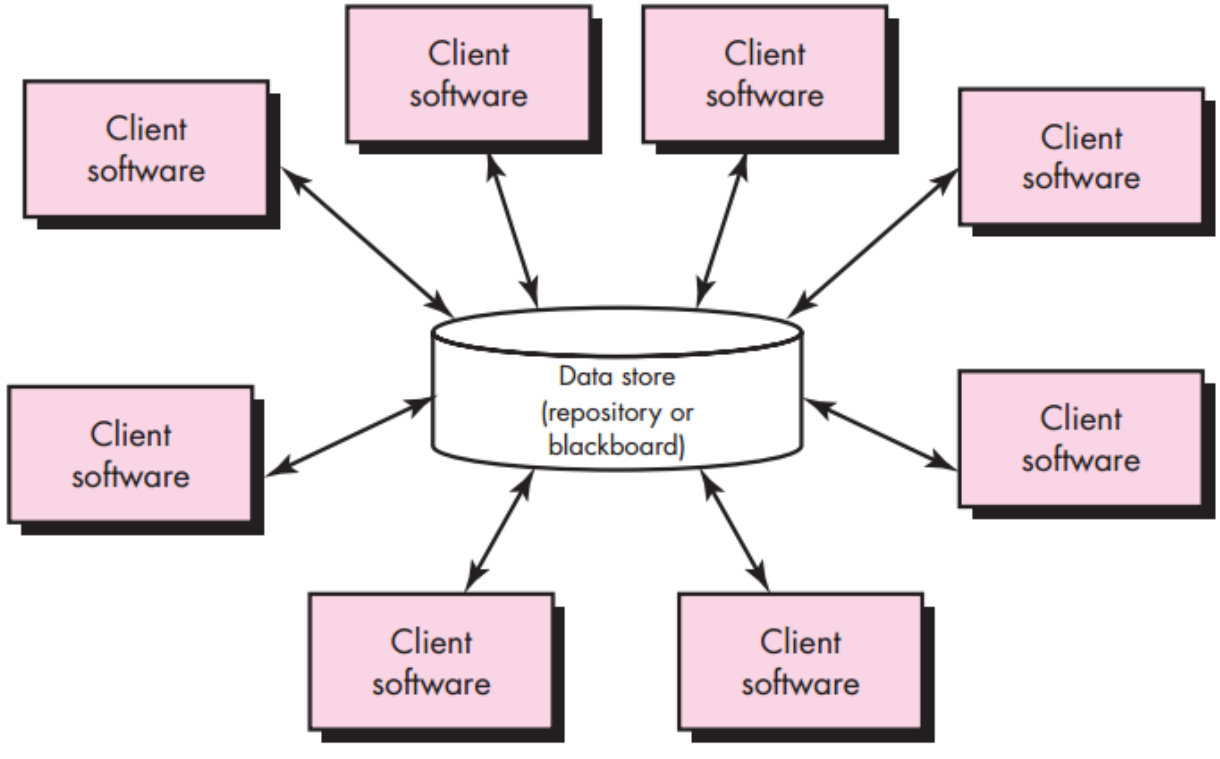
5.1.9 A Brief Taxonomy of Architectural Styles

1. Data-centered architectures

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

In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software.

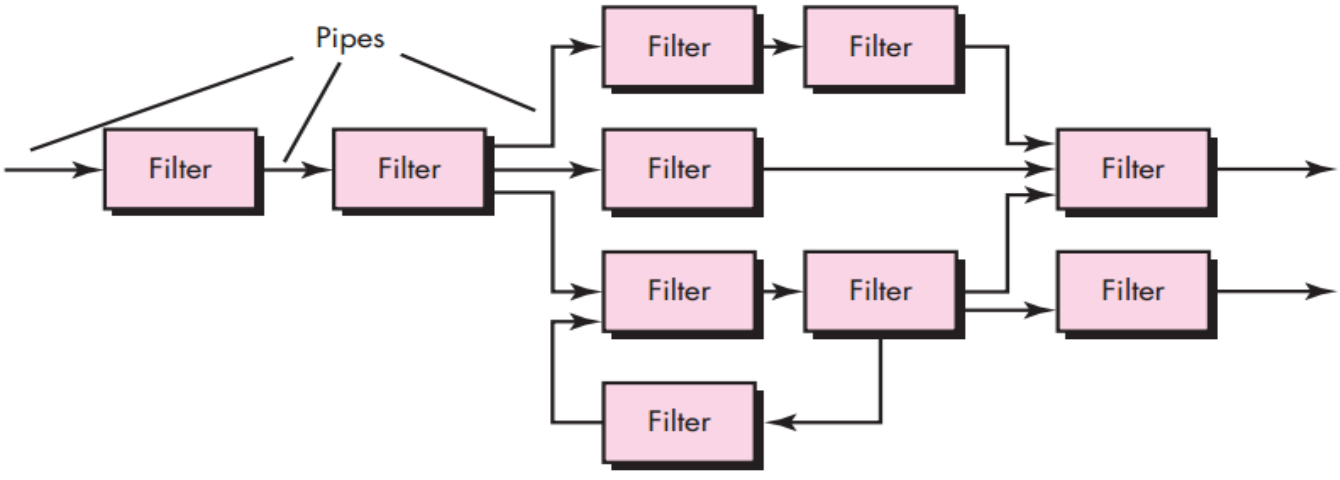
Data-centered architectures promote integrability and Client components can also independently execute processes.



2. 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 (Figure below) 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 workings of its neighboring filters.

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



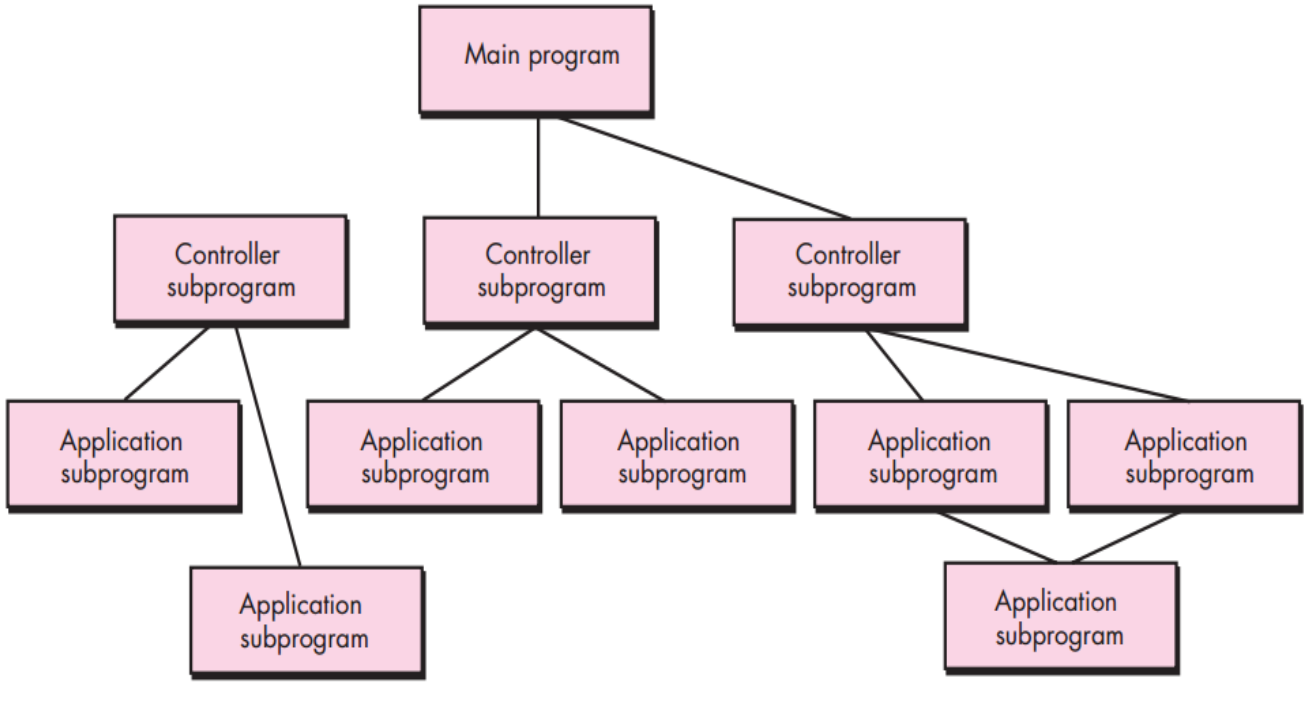
Pipes and filters

3. Call and return architectures

This architectural style enables to achieve a program structure that is relatively easy to modify and scale.

A number of sub styles exist within this category:

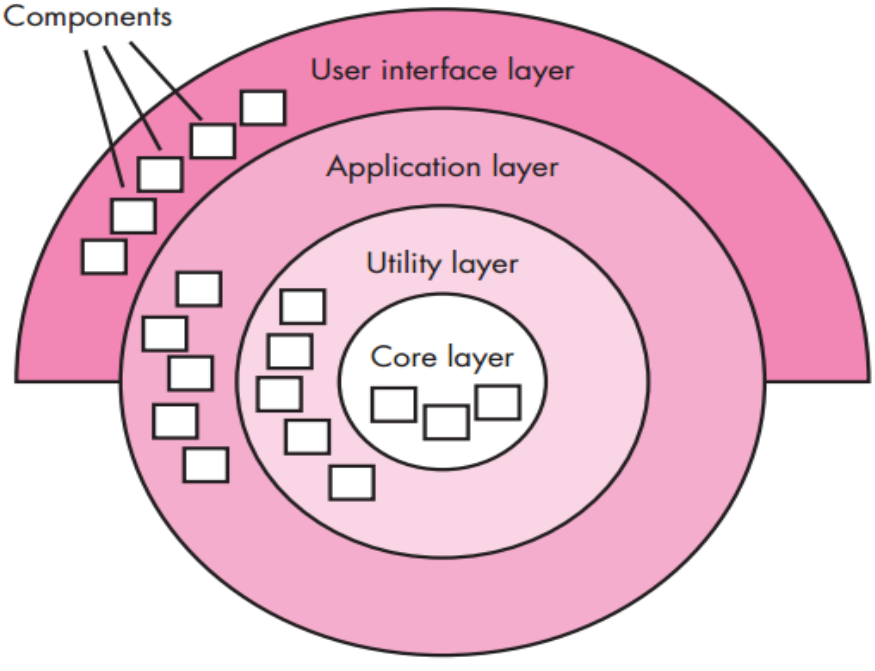
- Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Figure below illustrates an architecture of this type.
- Remote procedure calls architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.



4. Layered architectures

The basic structure of a layered architecture is illustrated in Figure below. 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.

These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style 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 (appropriate for most systems) can be combined with a data-centered architecture in many database applications.



5.2 Defining Software Architecture

What Is Software Architecture?

The software architecture of a program or computing system is a depiction of the system that aids in the understanding of how the system will behave.

Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams. The architecture is the primary carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision. Architecture is an artifact for early analysis to make sure that a design approach will yield an acceptable system. By building effective

architecture, you can identify design risks and mitigate them early in the development process.

Software architecture refers to the fundamental structures of a software system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system. Each structure comprises software elements, relations among them, and properties of both elements and relations, along with rationale for the introduction and configuration of each element. The architecture of a software system is a metaphor, analogous to the architecture of a building.

Software architecture is about making fundamental structural choices which are costly to change once implemented. Software architecture choices, also called architectural decisions, include specific structural options from possibilities in the design of software. For example, the systems that controlled the space shuttle launch vehicle had the requirement of being very fast and very reliable. Therefore, an appropriate real-time computing language would need to be chosen. Additionally, to satisfy the need for reliability the choice could be made to have multiple redundant and independently produced copies of the program, and to run these copies on independent hardware while cross-checking results.

Documenting software architecture facilitates communication between stakeholders, captures decisions about the architecture design, and allows reuse of design components between projects

5.2.1 Characteristics

Software architecture exhibits the following:

Multitude of stakeholders: software systems have to cater to a variety of stakeholders such as business managers, application owners, developers, end users and infrastructure operators. These stakeholders all have their own concerns with respect to the system. Balancing these concerns and demonstrating how they are addressed is part of designing the system. This implies that architecture involves dealing with a broad variety of concerns and stakeholders, and has a multidisciplinary nature. Software architect require non-technical's skills such as communication and negotiation competencies.

Separation of concerns: the established way for architects to reduce complexity is to separate the concerns that drive the design. Architecture documentation shows that all stakeholder concerns are addressed by modeling and describing the architecture from separate points of view associated with the various stakeholder concerns. These separate descriptions are called architectural views.

Quality-driven: classic software design approaches were driven by required functionality and the flow of data through the system, but the current insight is that the architecture of a software system is more closely related to its quality attributes

such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and other such –ilities. Stakeholder concerns often translate into requirements and constraints on these quality attributes, which are variously called non-functional requirements, extra-functional requirements, behavioral requirements, or quality attribute requirements.

Recurring styles: like building architecture, the software architecture discipline has developed standard ways to address recurring concerns. These "standard ways" are called by various names at various levels of abstraction. Common terms for recurring solutions are architectural style, principle, tactic, reference architecture and architectural pattern.

Conceptual integrity: a term introduced by Fred Brooks in The Mythical Man-Month to denote the idea that the architecture of a software system represents an overall vision of what it should do and how it should do it. This vision should be separated from its implementation. The architect assumes the role of "keeper of the vision", making sure that additions to the system are in line with the architecture, hence preserving conceptual integrity

5.3. Data Design

Data design translates the data objects defined in the analysis model into data structures that reside within the software. The attributes that describe the object, the relationships between data objects and their use within the program all influence the choice of data structures. At a higher level of abstraction, data design may lead to the definition of architecture for a database or a data warehouse.

- Data modeling is a prerequisite to good data design.
- Program data are encapsulated as a set of abstractions that are serviced by sub functions. The concepts of data encapsulation, information hiding, and data typing are used to create the data design.

Data design is the first design activity, which results in less complex, modular and efficient program structure. The information domain model developed during analysis phase is transformed into data structures needed for implementing the software. The data objects, attributes, and relationships depicted in entity relationship diagrams and the information stored in data dictionary provide a base for data design activity. During the data design process, data types are specified along with the integrity rules required for the data. For specifying and designing efficient data structures, some principles should be followed.

These **principles** are listed below.

- The data structures needed for implementing the software as well-as the operations that can be applied on them should be identified.

- A data dictionary should be developed to depict how different data objects interact with each other and what constraints are to be imposed on the elements of data structure.
- Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
- Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
- A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.
- Language used for developing the system should support abstract data types.

The **structure of data** can be viewed at three levels, namely,

- program component level,
- application level, and
- business level.

- At the program component level, the design of data structures and the algorithms required to manipulate them is necessary, if high-quality software is desired.
- At the application level, it is crucial to convert the data model into a database so that the specific business objectives of a system could be achieved.
- At the business level, the collection of information stored in different databases should be reorganized into data warehouse, which enables data mining that has an influential impact on the business.

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.

5.4 Architectural mapping using Data Flow Transform And Transaction Mapping

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 illustrate this approach, lets consider the SafeHomesecurity function. One element of the analysis

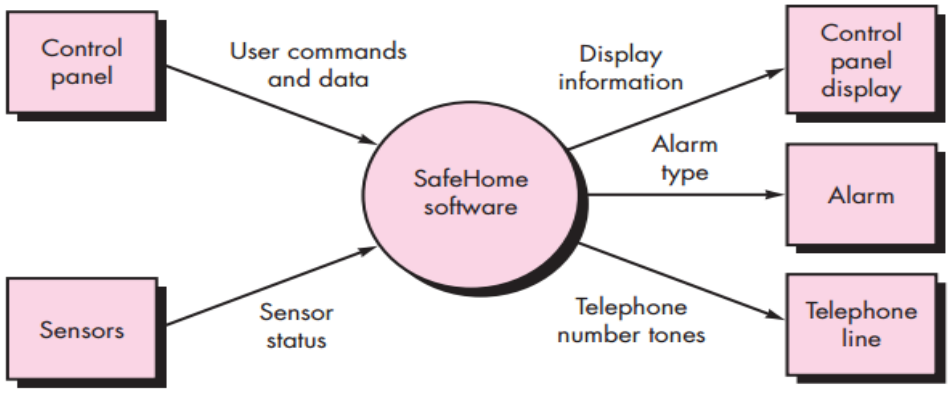
model is a set of data flow diagrams that describe information flow within the security function. To map these data flow diagrams into a software architecture, you would initiate the following design steps:

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

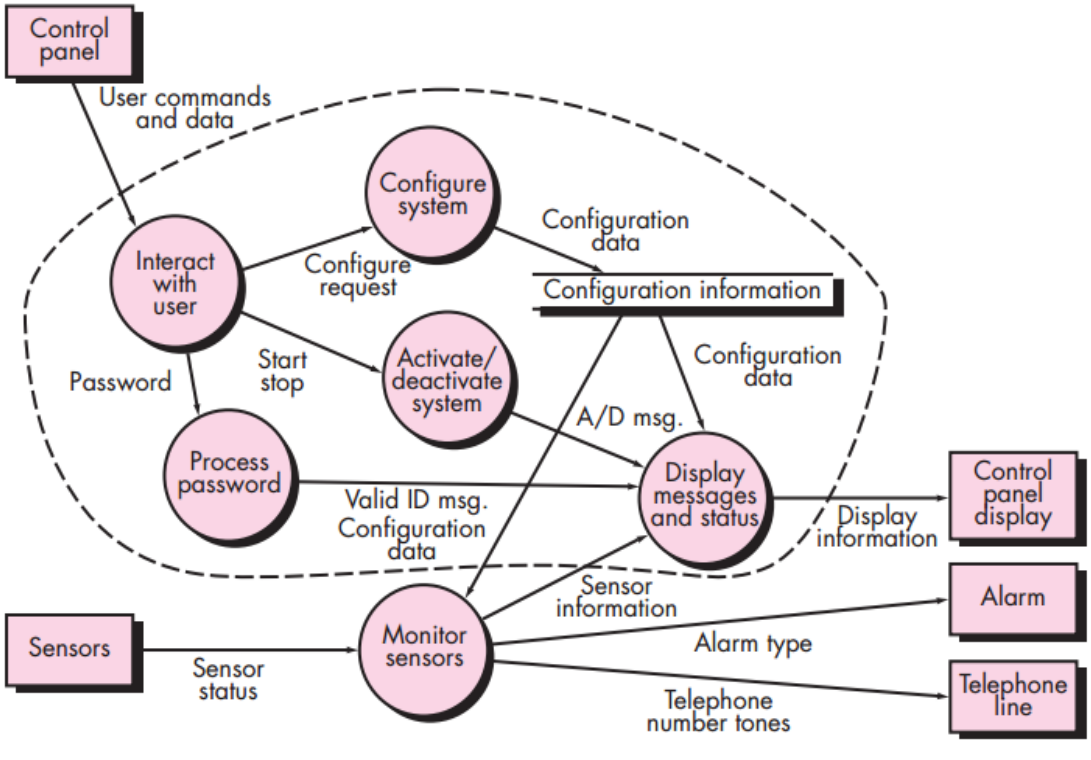
FIGURE

Context-level DFD for the *SafeHome* security function



FIGURE

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 (Figure below) is examined, and a level 3 data flow diagram is derived as shown in next Figure. 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 component in the SafeHomesoftware. 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.

FIGURE

Level 2 DFD that refines the *monitor sensors* transform

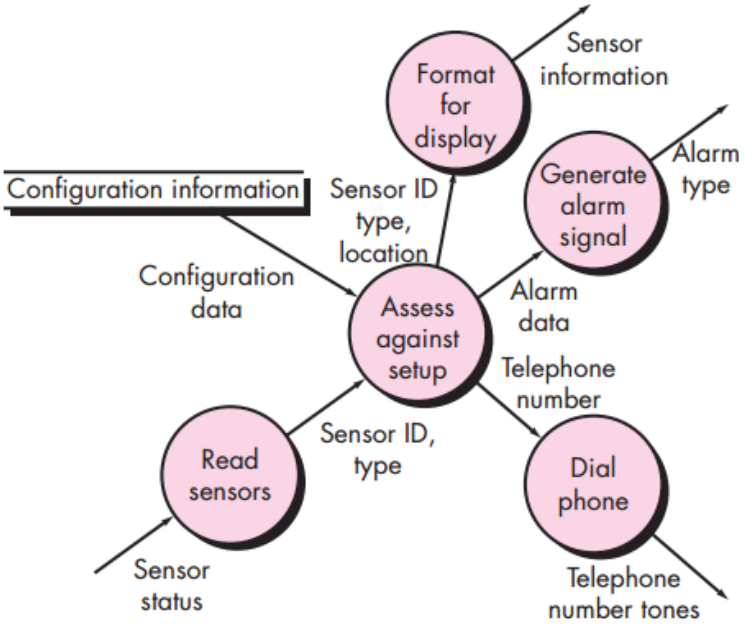
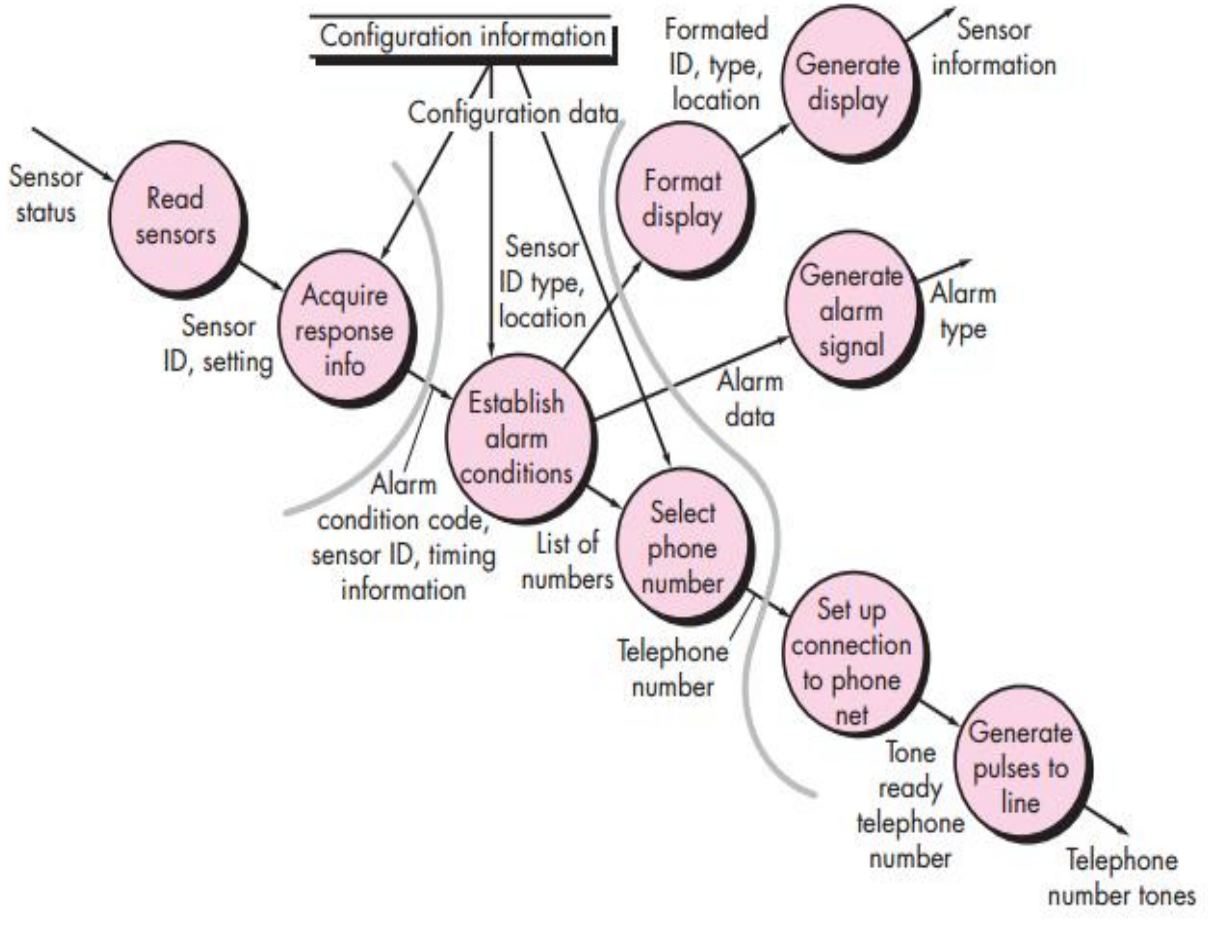


FIGURE Level 3 DFD for *monitor sensors* with flow boundaries



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

Evaluating the DFD (Figure Level 3), it is seen 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.

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. Incoming and outgoing flow boundaries are open to interpretation.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure Level 3. 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.

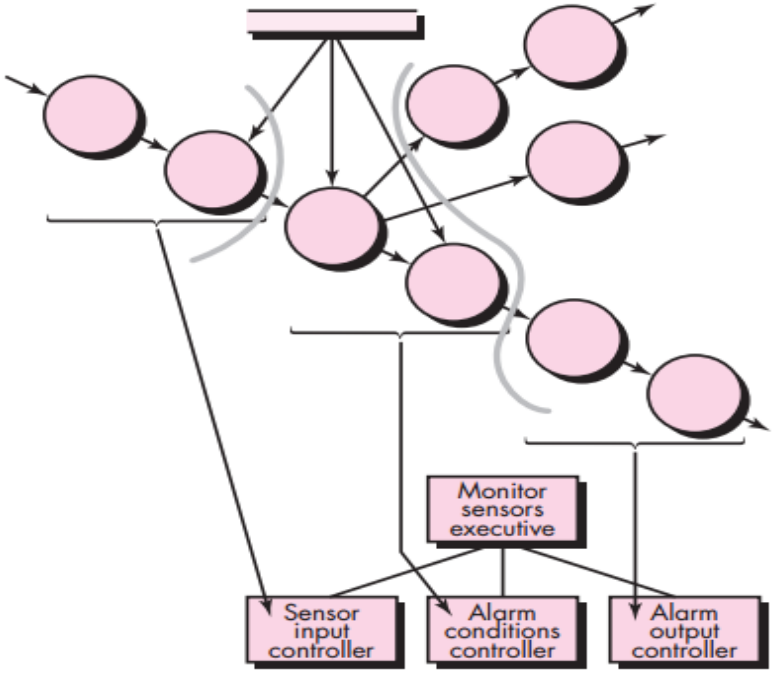
Step 5. Perform "first-level factoring."

The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top-level components perform decision making and lowlevel 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 below

FIGURE

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:

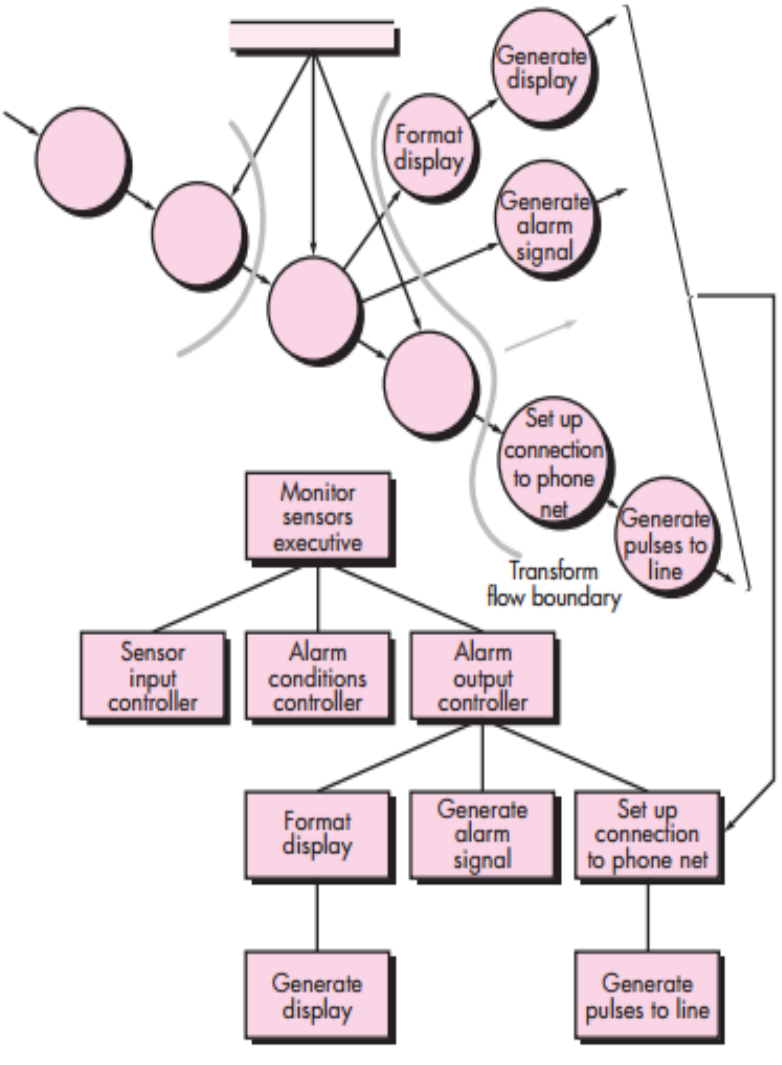
- An incoming information processing controller, called sensor input controller, coordinates receipt of all incoming data.
- A transform flow controller, called alarm conditions controller, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation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

Although a three-pronged structure is implied by Figure above, 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 functional independence 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 is illustrated in Figure below.

FIGURE
Second-level factoring for *monitor sensors*

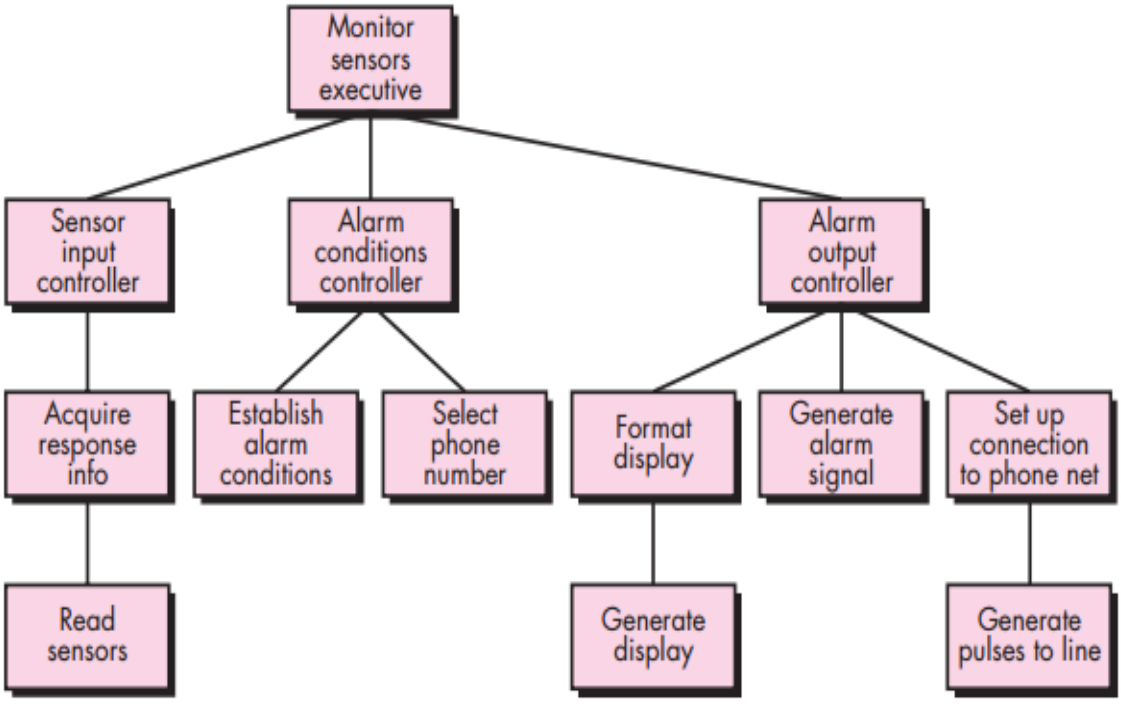


Although Figure above illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components. Practical considerations and measures of design quality dictate the outcome of second-level 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 below.

FIGURE

First-iteration structure for *monitor sensors*



The components mapped in the preceding manner and shown in Figure above represent an initial design of software architecture. Although components are named in a manner that implies function, a brief processing narrative (adapted from the process specification developed for a data transformation created during requirements modeling) should be written for each. The narrative describes the component interface, internal data structures, a functional narrative, and a brief discussion of restrictions and special features (e.g., file input-output, hardware dependent characteristics, special timing requirements).

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.

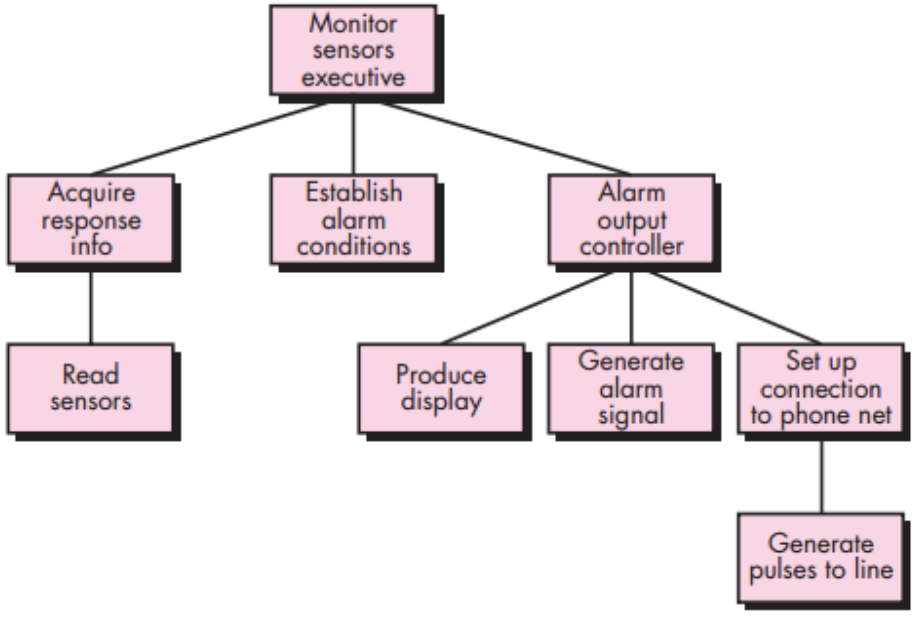
Software requirements coupled with human judgment is the final arbiter.

The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, it can be evaluated and refined software

architecture by viewing it as a whole. Modifications made at this time require little additional work, yet it can have a profound impact on software quality.

FIGURE

Refined program structure for *monitor sensors*



6. User Interface Design

(from Ian Sommerville)

- Designing effective interfaces for software systems
- Topics covered
  - User interface design principles
  - User interaction
  - Interface evaluation

The User Interface

- System users often judge a system by its interface rather than its functionality
- A poorly designed interface can cause a user to make catastrophic errors
- Poor user interface design is the reason why so many software systems are never used

Graphical User Interfaces

Most users of business systems interact with these systems through graphical interfaces although, in some cases, legacy text-based interfaces are still used

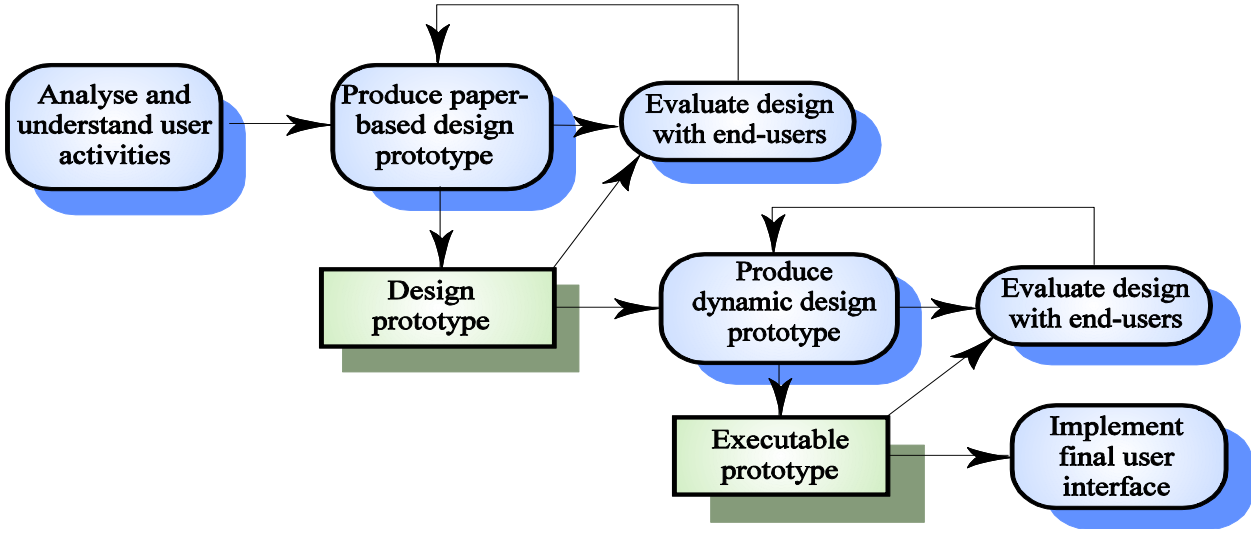
GUI Characteristics

Characteristic	Description
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files; on others, icons represent processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interest in a window.
Graphics	Graphical elements can be mixed with text on the same display.

GUI Advantages

- They are easy to learn and use.
  - Users without experience can learn to use the system quickly.
- The user may switch quickly from one task to another and can interact with several different applications.
  - Information remains visible in its own window when attention is switched.
- Fast, full-screen interaction is possible with immediate access to anywhere on the screen

User Interface Design Process



6.1 UI Design Principles

- UI design must take account of the needs, experience and capabilities of the system users
- Designers should be aware of people's physical and mental limitations (e.g. limited short-term memory) and should recognise that people make mistakes
- UI design principles underlie interface designs although not all principles are applicable to all designs

UI Design Principles

1. User familiarity
    - The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.
2. Consistency
    - The system should display an appropriate level of consistency. Commands and menus should have the same format; command punctuation should be similar, etc.
3. Minimal surprise
    - If a command operates in a known way, the user should be able to predict the operation of comparable commands
4. Recoverability
    - The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.
5. User guidance
    - Some user guidance such as help systems, on-line manuals, etc. should be supplied
6. User diversity
    - Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available

7. Real-time Software Design

(Ian Sommerville)

- Designing embedded software systems whose behaviour is subject to timing constraints
- Topics covered
    - 1. Real Time Systems design
    - 2. Real-time Executives
    - 3. Monitoring and Control Systems
    - 4. Data Acquisition Systems

7.1 Real-Time Systems

- Systems which monitor and control their environment

- Inevitably associated with hardware devices
  - Sensors: Collect data from the system environment
  - Actuators: Change (in some way) the system's environment
- Time is critical. Real-time systems MUST respond within specified times

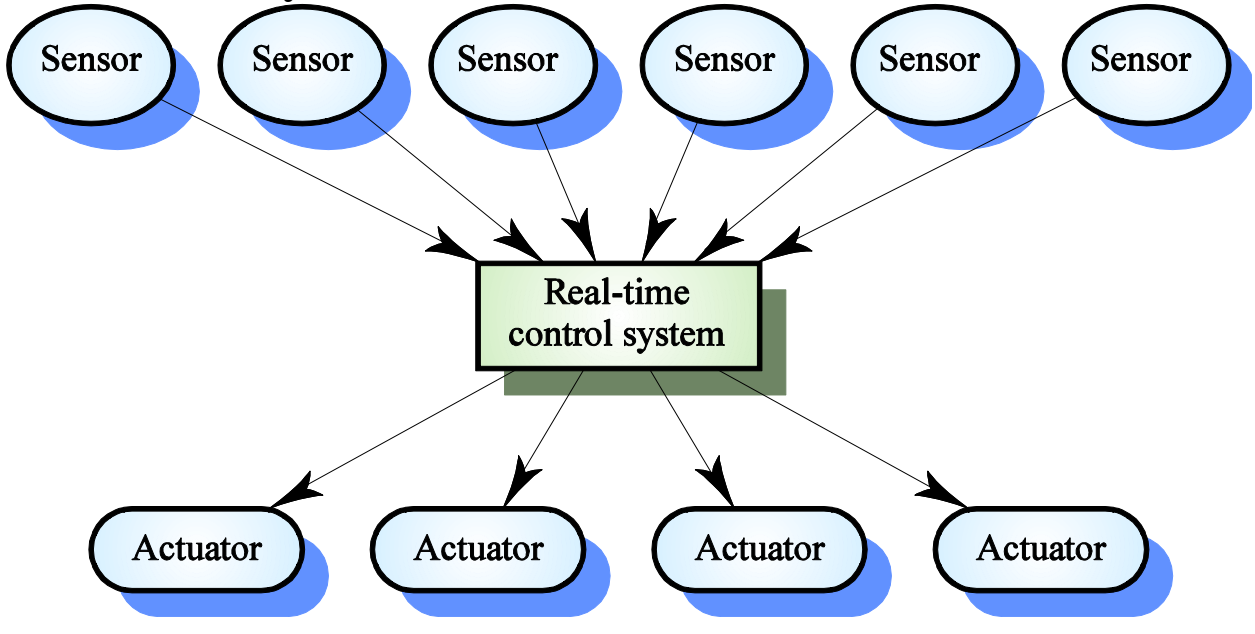
7.1.1 Definition

- A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced
- A 'soft' real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirements
- A 'hard' real-time system is a system whose operation is incorrect if results are not produced according to the timing specification

7.1.2 Stimulus/Response Systems

- Given a stimulus, the system must produce a response within a specified time
- Periodic Stimuli. Stimuli which occur at predictable time intervals
  - For example, a temperature sensor may be polled 10 times per second
- Aperiodic Stimuli. Stimuli which occur at unpredictable times
  - For example, a system power failure may trigger an interrupt which must be processed by the system

A real-time system model



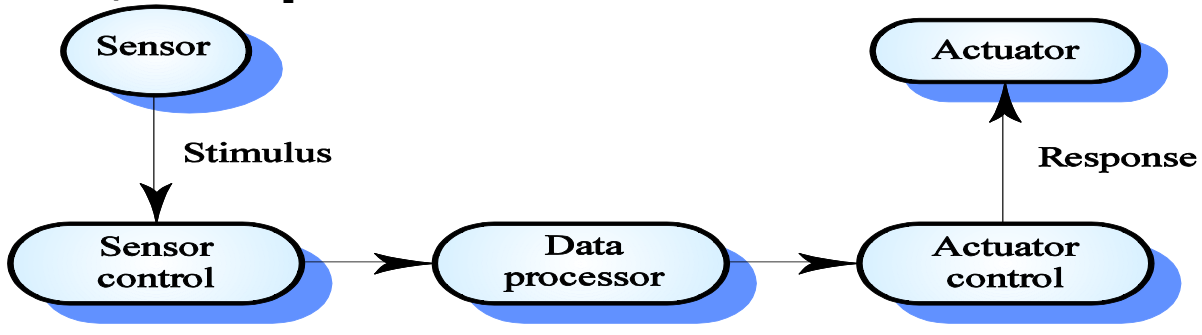
System Elements

- Sensors control processes
  - Collect information from sensors. May buffer information collected in response to a sensor stimulus
- Data processor

- Carries out processing of collected information and computes the system response

- Actuator control
  - Generates control signals for the actuator

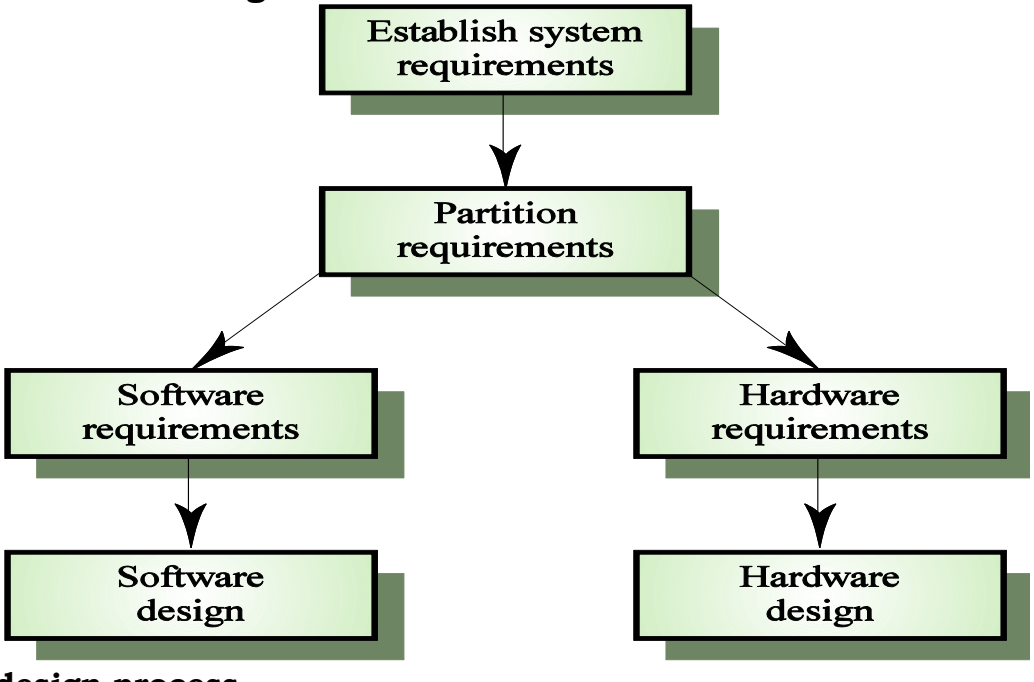
Sensor/actuator processes



7.2 Real Time System Design

- Design both the hardware and the software associated with system. Partition functions to either hardware or software
- Design decisions should be made on the basis on non-functional system requirements
- Hardware delivers better performance but potentially longer development and less scope for change

Hardware and Software Design



R-T systems design process

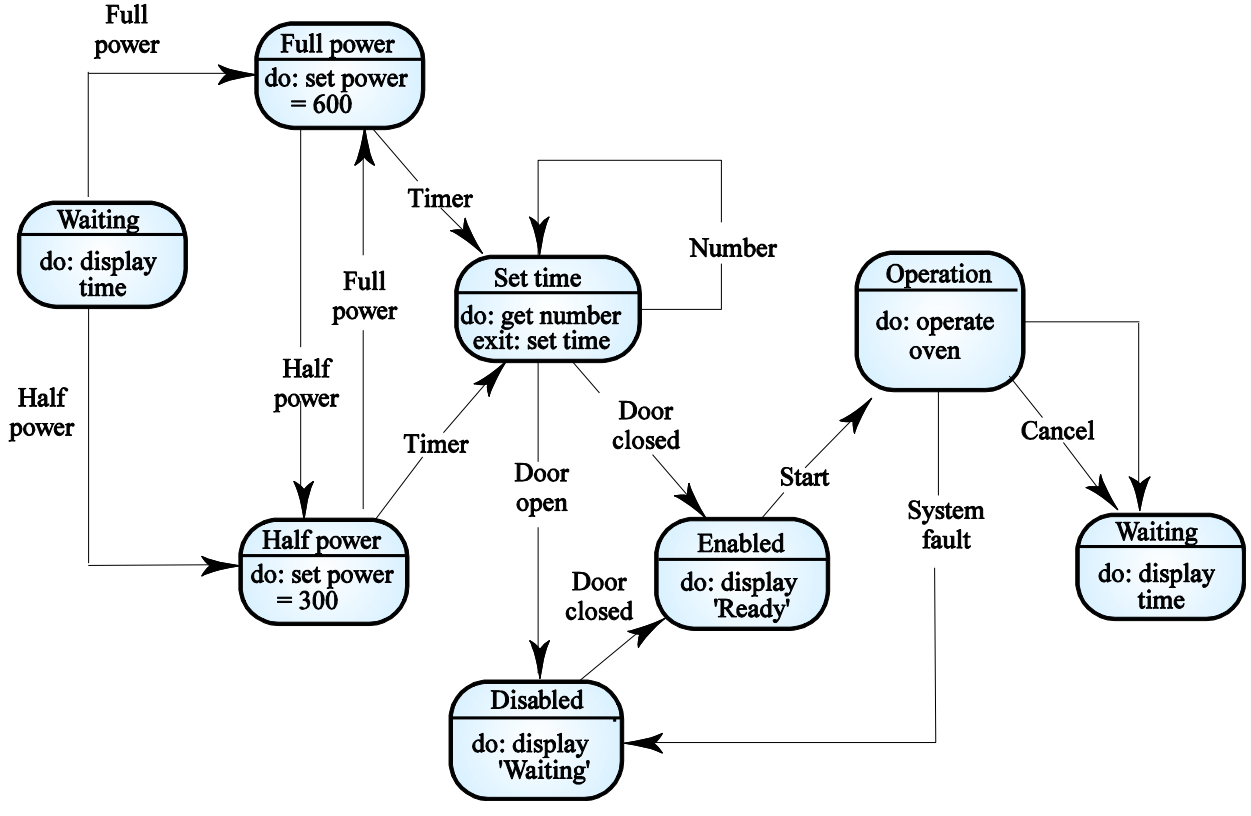
1. Identify the stimuli to be processed and the required responses to these stimuli
2. For each stimulus and response, identify the timing constraints

3. Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response
4. Design algorithms to process each class of stimulus and response. These must meet the given timing requirements
5. Design a scheduling system which will ensure that processes are started in time to meet their deadlines
6. Integrate using a real-time executive or operating system

State machine modelling

- The effect of a stimulus in a real-time system may trigger a transition from one state to another.
- Finite state machines can be used for modelling real-time systems.
- However, FSM models lack structure. Even simple systems can have a complex model.
- The UML includes notations for defining state machine models

Microwave oven state machine



7.3 Real-time executives

- Real-time executives are specialised operating systems which manage the processes in the RTS
- Responsible for process management and resource (processor and memory) allocation

- May be based on a standard RTE kernel which is used unchanged or modified for a particular application

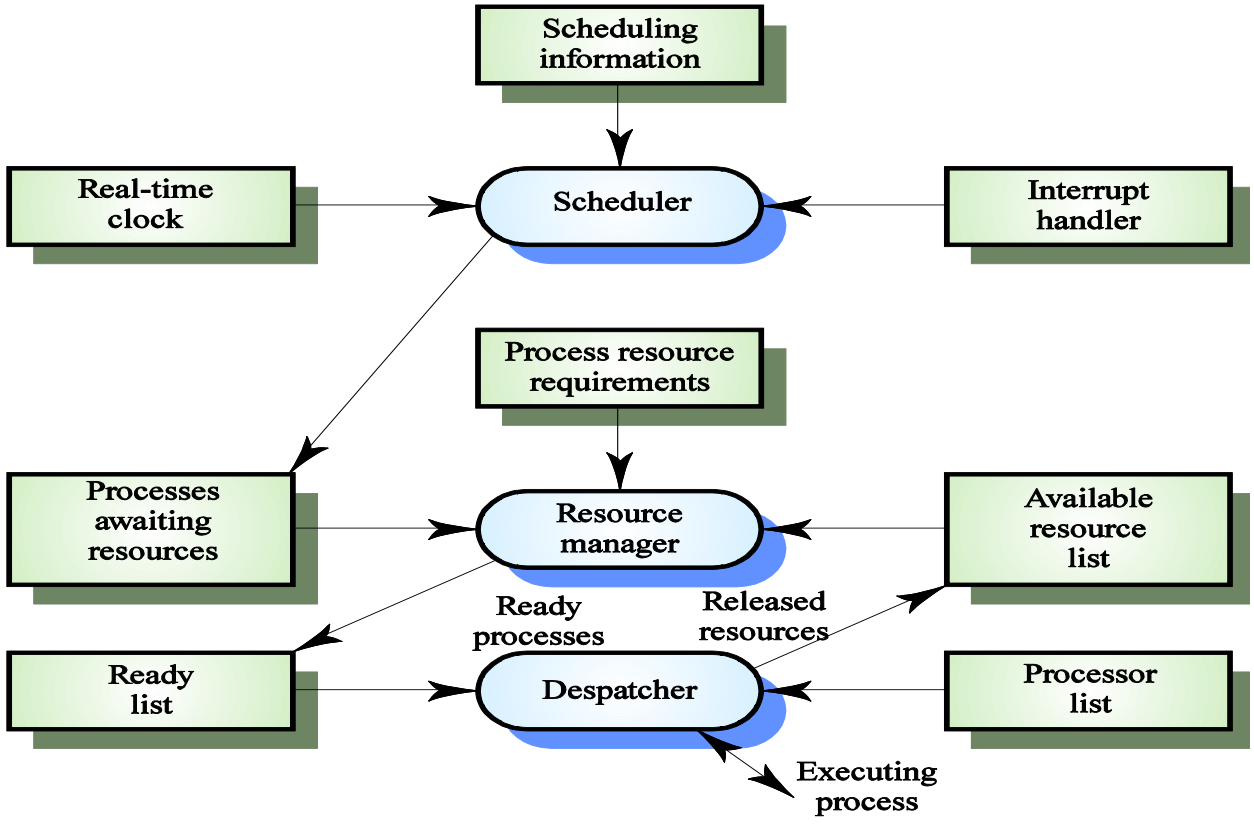
Executive components

1. Real-time clock
   - Provides information for process scheduling.
2. Interrupt handler
   - Manages aperiodic requests for service.
3. Scheduler
   - Chooses the next process to be run.
4. Resource manager
   - Allocates memory and processor resources.
5. Despatcher
   - Starts process execution.

Non-stop System Components

1. Configuration Manager
   - Responsible for the dynamic reconfiguration of the system software and hardware. Hardware modules may be replaced and software upgraded without stopping the systems
2. Fault Manager
   - Responsible for detecting software and hardware faults and taking appropriate actions (e.g. switching to backup disks) to ensure that the system continues in operation

Real-time executive components



7.4 Monitoring and Control Systems

- Important class of real-time systems
- Continuously check sensors and take actions depending on sensor values
- Monitoring systems examine sensors and report their results
- Control systems take sensor values and control hardware actuators

7.4.1 Burglar Alarm System

- A system is required to monitor sensors on doors and windows to detect the presence of intruders in a building
- When a sensor indicates a break-in, the system switches on lights around the area and calls police automatically
- The system should include provision for operation without a mains power supply

7.4.2 Burglar Alarm System

- Sensors
  - Movement detectors, window sensors, door sensors.
  - 50 window sensors, 30 door sensors and 200 movement detectors
  - Voltage drop sensor
- Actions
  - When an intruder is detected, police are called automatically.
  - Lights are switched on in rooms with active sensors.
  - An audible alarm is switched on.
  - The system switches automatically to backup power when a voltage drop is detected.

7.4.3 The R-T System Design Process

- Identify stimuli and associated responses
- Define the timing constraints associated with each stimulus and response
- Allocate system functions to concurrent processes
- Design algorithms for stimulus processing and response generation
- Design a scheduling system which ensures that processes will always be scheduled to meet their deadlines

7.4.4 Stimuli to be processed

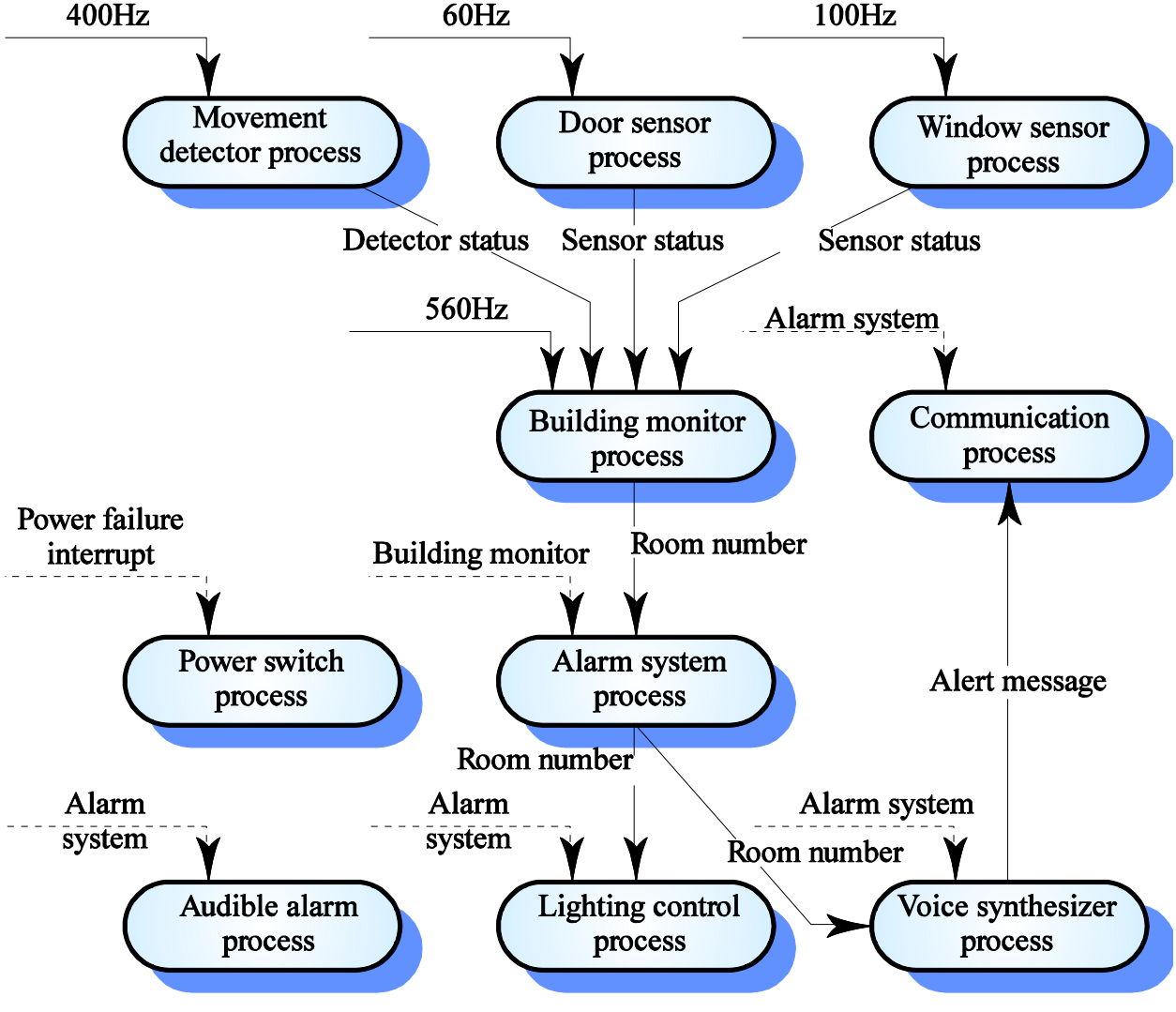
- Power failure
  - Generated aperiodically by a circuit monitor. When received, the system must switch to backup power within 50 ms
- Intruder alarm
  - Stimulus generated by system sensors. Response is to call the police, switch on building lights and the audible alarm

Timing Requirements

Stimulus/Response	Timing requirements
Power fail interrupt	The switch to backup power must be completed within a deadline of 50 ms.
Door alarm	Each door alarm should be polled twice per second.

Window alarm	Each window alarm should be polled twice per second.
Movement detector	Each movement detector should be polled twice per second.
Audible alarm	The audible alarm should be switched on within 1/2 second of an alarm being raised by a sensor.
Lights switch	The lights should be switched on within 1/2 second of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2 seconds of an alarm being raised by a sensor.
Voice synthesiser	A synthesised message should be available within 4 seconds of an alarm being raised by a sensor.

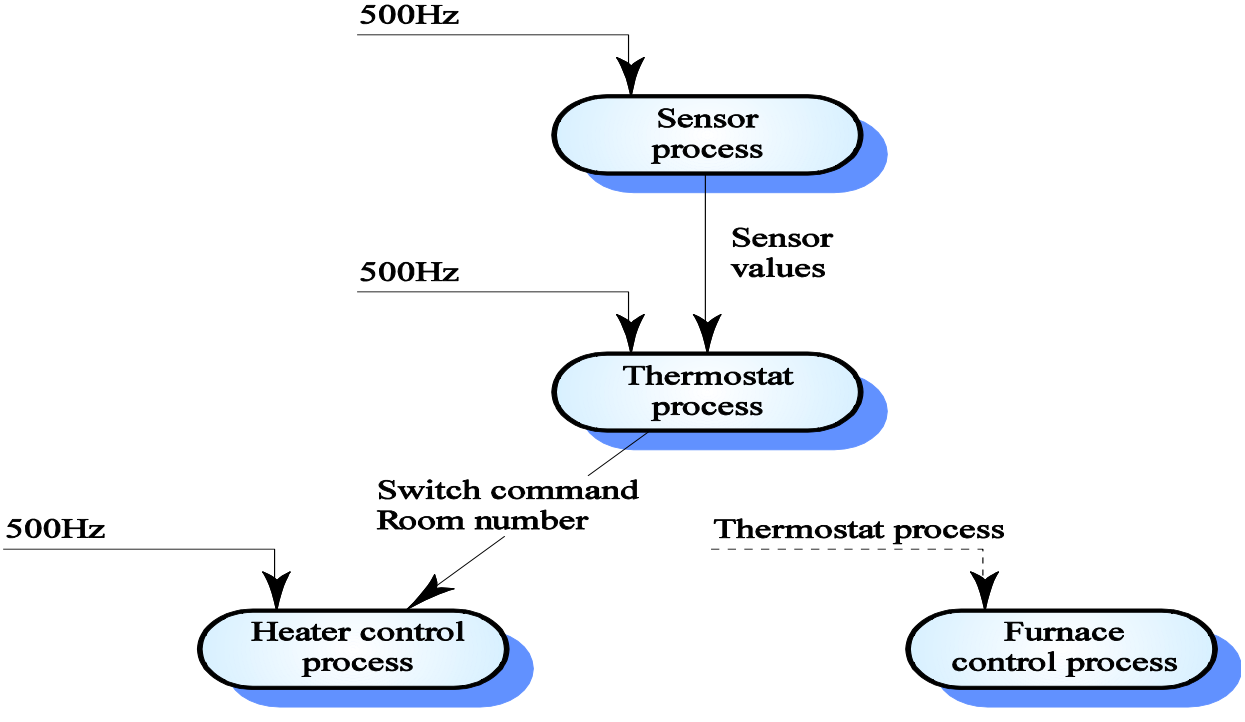
Process Architecture



7.4.5 Control Systems

- A burglar alarm system is primarily a monitoring system. It collects data from sensors but no real-time actuator control
- Control systems are similar but, in response to sensor values, the system sends control signals to actuators
- An example of a monitoring and control system is a system which monitors temperature and switches heaters on and off

A temperature control system



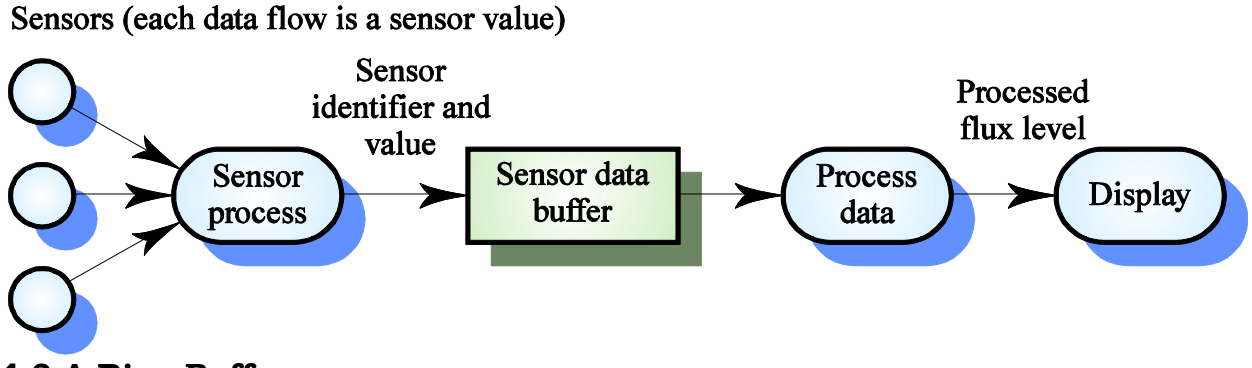
7.5 Data Acquisition Systems

- Collect data from sensors for subsequent processing and analysis.
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing
  e.g. collecting information about an explosion.
- Circular or ring buffers are a mechanism for smoothing speed differences.

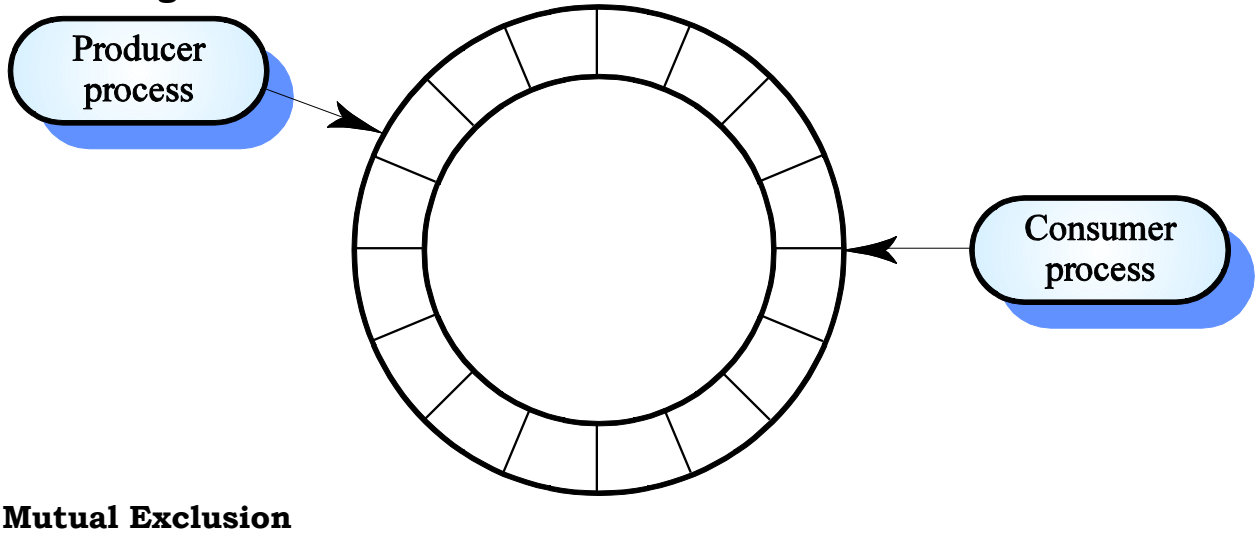
Reactor Data Collection

- A system collects data from a set of sensors monitoring the neutron flux from a nuclear reactor.
- Flux data is placed in a ring buffer for later processing.
- The ring buffer is itself implemented as a concurrent process so that the collection and processing processes may be synchronized.

Reactor Flux Monitoring



4.3 A Ring Buffer



Mutual Exclusion

- Producer processes collect data and add it to the buffer. Consumer processes take data from the buffer and make elements available
- Producer and consumer processes must be mutually excluded from accessing the same element.
- The buffer must stop producer processes adding information to a full buffer and consumer processes trying to take information from an empty buffer.

Key points

- Real-time system correctness depends not just on what the system does but also on how fast it reacts
- A general RT system model involves associating processes with sensors and actuators
- Real-time systems architectures are usually designed as a number of concurrent processes

8. Software Configuration Management

FAQs

1. What is it?

When you build computer software, change happens. And because it happens, you need to manage it effectively. Software configuration management (SCM), also called change management, is a set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

2. Who does it?

Everyone involved in the software process is involved with change management to some extent, but specialized support positions are sometimes created to manage the SCM process.

3. Why is it important?

If you don't control change, it controls you. And that's never good.
It's very easy for a stream of uncontrolled changes to turn a well-run software project into chaos. As a consequence, software quality suffers and delivery is delayed. For that reason, change management is an essential part of quality management.

4. What are the steps?

Because many work products are produced when software is built, each must be uniquely identified. Once this is accomplished, mechanisms for version and change control can be established. To ensure that quality is maintained as changes are made, the process is audited; and to ensure that those with a need to know are informed about changes, reporting is conducted.

5. What is the work product?

A Software Configuration Management Plan defines the project strategy for change management. In addition, when formal SCM is invoked, the change control process produces software change requests, reports, and engineering change orders.

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to

(1) identify change,
(2) control change,
(3) ensure that change is being properly implemented, and
(4) report changes to others who may have an interest

It is important to make a clear distinction between software support and software configuration management. Support is a set of software engineering activities that

occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that are initiated when a software engineering project begins and terminates only when the software is taken out of operation.

The output of the software process is information that may be divided into three broad categories:

(1) computer programs (both source level and executable forms),
(2) work products that describe the computer programs (targeted at various stakeholders), and
(3) data or content (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a ***software configuration***.

Elements of a Configuration Management System

Four important elements that should exist when a configuration management system is developed

- Component elements—a set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- Process elements—a collection of actions and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.
- Construction elements—a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- Human elements—a set of tools and process features (encompassing other CM elements) used by the software team to implement effective SCM.

Baselines

A baseline is a software configuration management concept that helps to control change without seriously impeding justifiable change. The IEEE defines a baseline as:

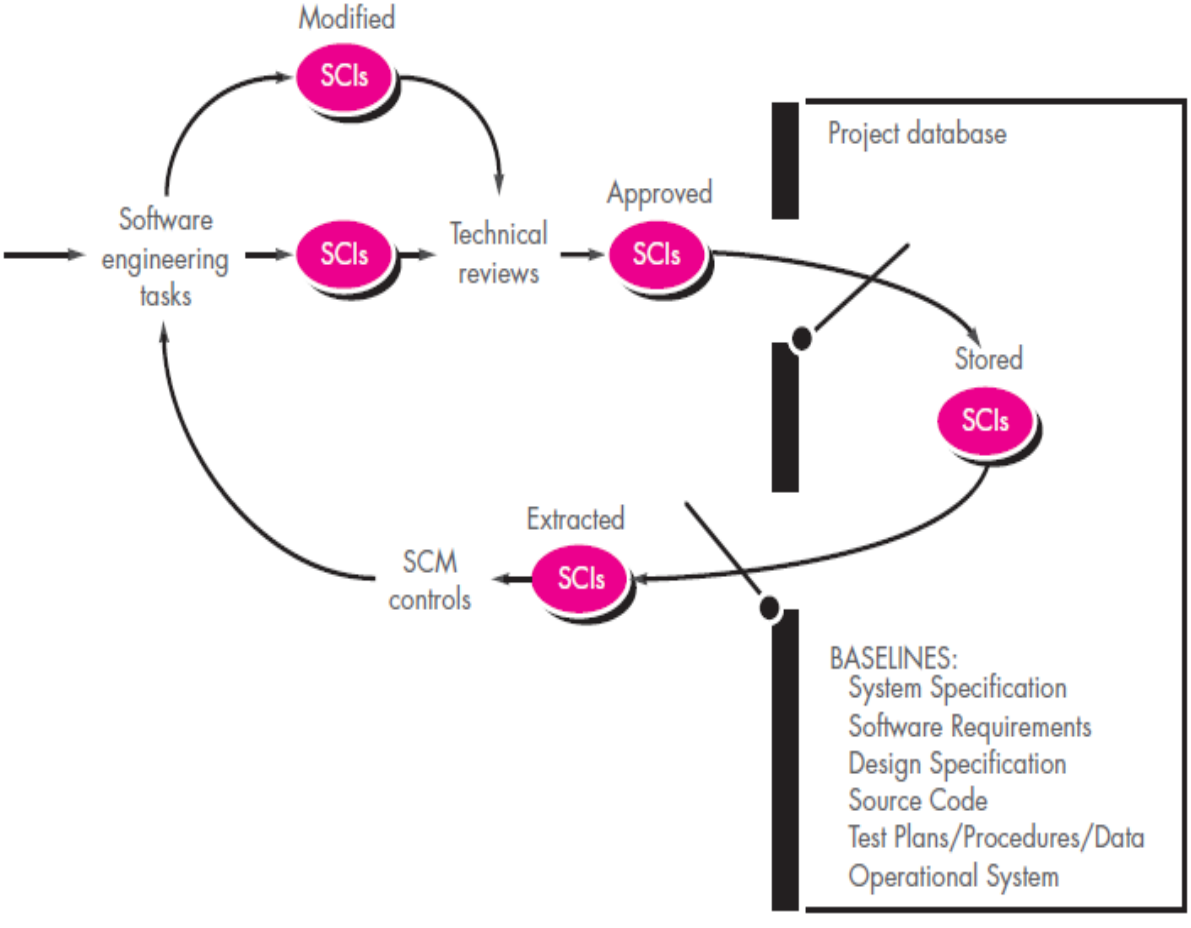
"A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures."

Before a software configuration item becomes a baseline, changes may be made quickly and informally. However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

In the context of software engineering, a baseline is a milestone in the development of software. A baseline is marked by the delivery of one or more software configuration items that have been approved as a consequence of a technical review.

For example, the elements of a design model have been documented and reviewed. Errors are found and corrected. Once all parts of the model have been reviewed, corrected, and then approved, the design model becomes a baseline. Further changes to the program architecture (documented in the design model) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure below

Fig: Baseline SCIs and the project database



Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a project database (also called a project library or software repository).

The SCM system has the following **advantages**:

- Reduced redundant work.
- Effective management of simultaneous updates.
- Avoids configuration-related problems.
- Facilitates team coordination.
- Helps in building management; managing tools used in builds.
- Defect tracking: It ensures that every defect has traceability back to its source.

8.1 Need for SCM

There are many needs of SCM. However, they can be grouped into these categories:

- Efficiency
- Reproducibility
- Traceability
- Configuration Controls

1. Efficiency

Developers can focus on writing code that meets their customer requirements.

Too many times, developers are side-tracked with build or configuration problems which distracts them from their primary function, writing solid applications.

The efficiencies gained from an Software Configuration Management team creating and maintaining the automated software builds and deployment automation greatly frees the developers to do what they do best.

- Software Builds are Standardized, Repeatable and Managed
- Individual Development Teams do not have to Define/Redefine Build Process
- Build Results are Immediately Published to Build Dashboard
- Ensure Rapid Resolution to all Build Issues
- Parallel Builds and Development for Faster Development Iterations
- Configurations are Known and Published
- Fully Automated Installation Procedures
- Encapsulate Run-time Server Configurations
- Reduce Development Cycle

2. Reproducibility

Reproducibility ensures that the output of the software build process produces the same result every time.

It eliminates the guess work

- Repeatable Software Builds
- Confidence that the Build Results are Valid
- Produces Evidence the Build Succeeded
- Creates Repeatable Installation Process

3. Trace Ability

Trace ability ensures that an organization can track a change from source code all the way to installed software product.

- All Source Files are Labeled
- All Build Artifacts are Versioned
- Test Environments are Controlled
- Create Build and Installation Logs

Software Configuration Management Configuration Controls

Controlling Configurations enables organizations to ensure that they know precisely what is in a new build and what changes are being tested.

Controls Include:
- Change Control Processes
- Build of Materials
- Proper Defect Tracking Procedures
- Audit-able Build and Install Logs

The benefits are many, but the overwhelming reason to use Software Configuration Management on your development projects is...***The increased likely hood of the project succeeding on time and within budget.***

8.2 The SCM Process

The software configuration management process defines a series of tasks that have four primary objectives:
(1) to identify all items that collectively define the software configuration,
(2) to manage changes to one or more of these items,
(3) to facilitate the construction of different versions of an application, and
(4) to ensure that software quality is maintained as the configuration evolves over time.

Referring to the figure below, SCM tasks can viewed as concentric layers. SCIs flow outward through these layers throughout their useful life, ultimately becoming part of the software configuration of one or more versions of an application or system.

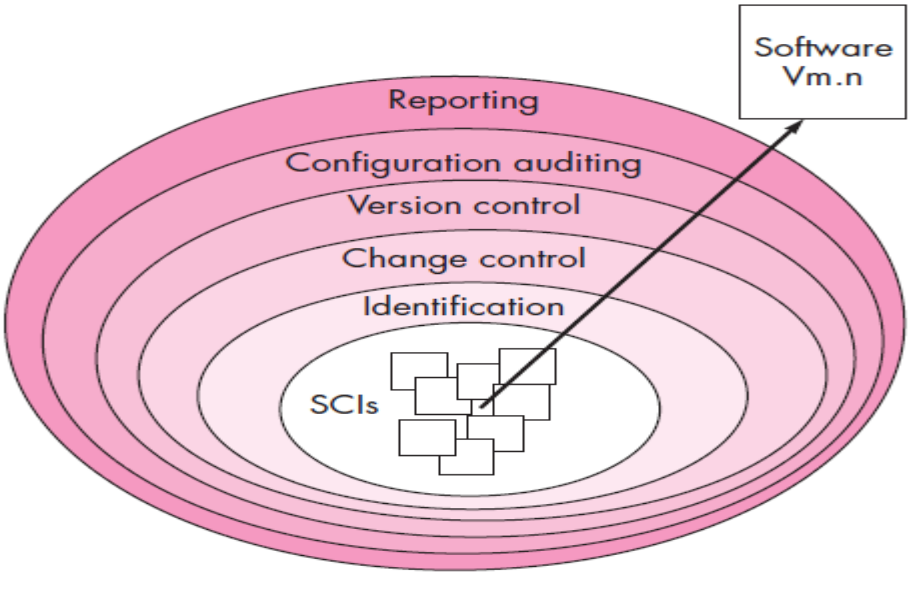


Figure: Layers of SVM Process

9. Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process.

A version control system implements or is directly integrated with four major capabilities:

(1) a project database (repository) that stores all relevant configuration objects,
(2) a version management capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions),
(3) a make facility that enables you to collect all relevant configuration objects and construct a specific version of the software.

In addition, version control and change control systems often implement an issues tracking (also called bug tracking) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

A number of version control systems establish a *change set*—a collection of all changes (to some baseline configuration) that are required to create a specific version of the software

Software tools used for Version control

SOFTWARE TOOLS

The Concurrent Versions System (CVS)

The use of tools to achieve version control is essential for effective change management. The *Concurrent Versions System* (CVS) is a widely used tool for version control. Originally designed for source code, but useful for any text-based file, the CVS system (1) establishes a simple repository, (2) maintains all versions of a file in a single named file by storing only the differences between progressive versions of the original file, and (3) protects against simultaneous changes to a file by establishing different directories for each developer, thus insulating one from another. CVS merges changes when each developer completes her work.

It is important to note that CVS is not a "build" system; that is, it does not construct a specific version of the software. Other tools (e.g., *Makefile*) must be integrated with CVS to accomplish this. CVS does not implement a change control process (e.g., change requests, change reports, bug tracking).

Even with these limitations, CVS "is a dominant open-source network-transparent version control system [that] is useful for everyone from individual developers to large, distributed teams" [CVS07]. Its client-server architecture allows users to access files via Internet connections, and its open-source philosophy makes it available on most popular platforms.

CVS is available at no cost for Windows, Mac OS, LINUX, and UNIX environments. See [CVS07] for further details.

10. Software Configuration Items SCIs

A software configuration item as information that is created as part of the software engineering process a SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically, an SCI is all or part of a work product (e.g., a document, an entire suite of test cases, or a named program component).

SCIs are organized to form configuration objects that may be catalogued in the project database with a single name. A configuration object has a name, attributes, and is "connected" to other objects by relationships.

Referring to Figure below, the configuration objects are,

- DesignSpecification,
- DataModel,
- ComponentN,
- SourceCode, and
- TestSpecification are each defined separately.

However, each of the objects is related to the others as shown by the arrows.

- A curved arrow indicates a compositional relation. That is, DataModel and ComponentN are part of the object DesignSpecification.
- A double-headed straight arrow indicates an interrelationship. If a change were made to the SourceCode object, the interrelationships enable to determine what other objects (and SCIs) might be affected

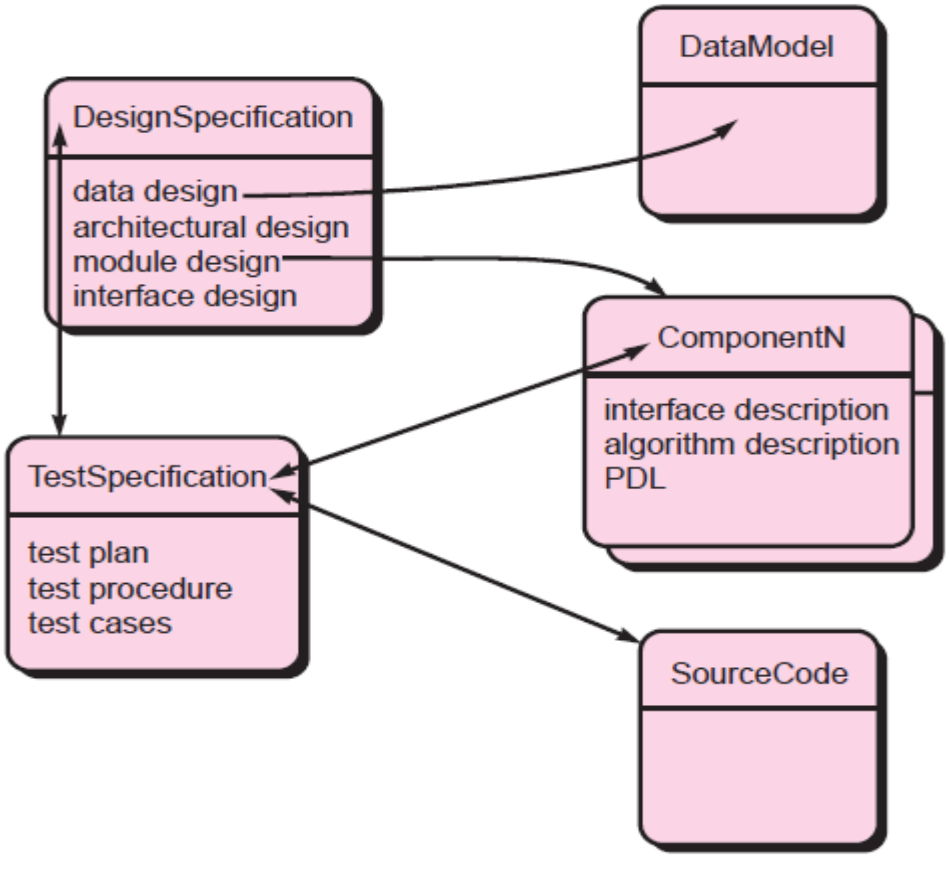


Figure: Configuration Objects (SCIs)

The SCM Repository

SCIs are maintained in a project database or repository. Webster's Dictionary defines the word repository as "anything or person thought of as a center of accumulation or storage."

Today, the repository is a "thing"— a database that acts as the center for both accumulation and storage of software engineering information. The role of the person (the software engineer) is to interact with the repository using tools that are integrated with it.

The Role of the Repository

The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration. In addition, the SCM repository provides a hub for the integration of software tools, is central to the flow of the software process, and can enforce uniform structure and format for software engineering work products.

A repository that **serves** a software engineering team should also

(1) integrate with or directly support process management functions,
(2) support specific rules that govern the SCM function and the data maintained within the repository,
(3) provide an interface to other software engineering tools, and
(4) accommodate storage of sophisticated data objects (e.g., text, graphics, video, audio).

Figure : Content of the SCI repository

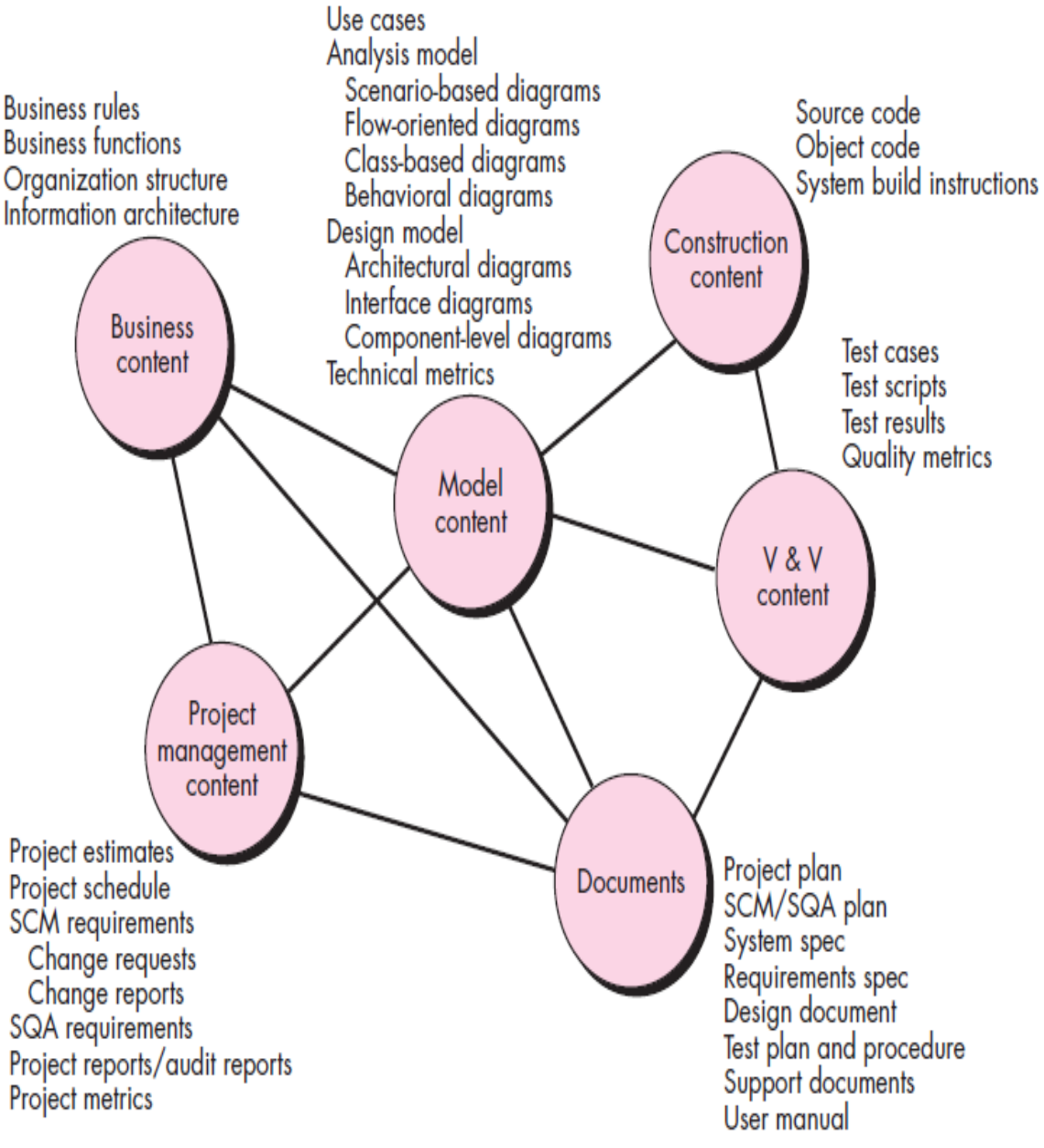


Figure : Content of the SCI repository