**Unit 7**

**Design Concepts and Principles**

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

Design is the place where software quality is established. The design process moves from a “big picture” view of software to a more narrow view that defines the detail required to implement a system. The process begins by focusing on architecture. Subsystems are defined; communication mechanisms among subsystems are established; components are identified, and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed.

Three characteristics that serve as a guide for the evaluation of a good design:

* The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.
* The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
* The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

**Generic task set for Design**

1. Examine the information domain model, and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture: Be certain that each subsystem is functionally cohesive. Design subsystem interfaces. Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components: Translate analysis class description into a design class. Check each design class against design criteria; consider inheritance issues. Define methods and messages associated with each design class. Evaluate and select design patterns for a design class or a subsystem. Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface: Review results of task analysis. Specify action sequence based on user scenarios. Create behavioral model of the interface. Define interface objects, control mechanisms. Review the interface design and revise as required.
7. Conduct component-level design. Specify all algorithms at a relatively low level of abstraction. Refine the interface of each component. Define component-level data structures. Review each component and correct all errors uncovered.
8. Develop a deployment model.

**Design concepts**

**Abstraction**

Software engineering posses many levels of abstraction. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. In any levels of abstraction we work with two abstractions: procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. A data abstraction is a named collection of data that describes a data object. For example, in the context of the procedural abstraction open, we can define a data abstraction called door.

**Architecture**

Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. One goal of software design is to derive an architectural rendering of a system. Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

* **Structural properties**. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
* **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
* **Families of related systems.** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

* **Structural models** represent architecture as an organized collection of program components.
* **Framework models** increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
* **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.
* **Process models** focus on the design of the business or technical process that the system must accommodate.
* **Functional models** can be used to represent the functional hierarchy of a system.

**Patterns**

In object-oriented idioms, the pattern is a problem solving technique by assigning some or specific responsibilities on an object. For example:

Pattern Name: Information Expert

Solution Assign a responsibility to the class that has the information needed to fulfill it

Problem It Solves What is a basic principle by which to assign

The intent of each design pattern is to provide a description that enables a designer to determine

* Whether the pattern is applicable to the current work,
* Whether the pattern can be reused (hence, saving design time), and
* Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

**Separation of Concerns**

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller and therefore more manageable pieces, a problem takes less effort and time to solve. Separation of concerns is expressed in other related design concepts: modularity, aspects, functional independence, and refinement.

**Modularity**

Modularity is the single attribute of software that allows a program to be intellectually manageable. Software is divided into separately named and addressable components, sometimes called modules, which are integrated to satisfy problem requirements. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

**Information Hiding**

The concept of modularity leads you to a fundamental question: “How do I decompose a software solution to obtain the best set of modules?” 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 (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

**Functional Independence**

Functional independence is a key to good design, and design is the key to software quality. The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding.

Functional independence is achieved by developing modules with “single minded” function and an “aversion” to excessive interaction with other modules. Independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.

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

* **High Cohesion**: High cohesion means assigning only one or less responsibility to an object so that it can be re-useable in many other systems also. Also, assigning some specific job to any system helps for efficiency of work increase.
* **Low Coupling:** Low coupling means less number of relations to other object (less dependency) that helps to isolate the object as re-usable component. Strongly coupled classes are undesirable; they suffer from the problems of: “hard to comprehend, reuse, maintain and change”.

**Refinement**

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. 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. Abstraction and refinement are complementary concepts. Abstraction enables us to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details. Refinement helps us to reveal low-level details as design progresses. Both concepts allow us to create a complete design model as the design evolves.

**Refactoring**

An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

**Data Design or Architecting**

Data design 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 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.

**Architectural Design**

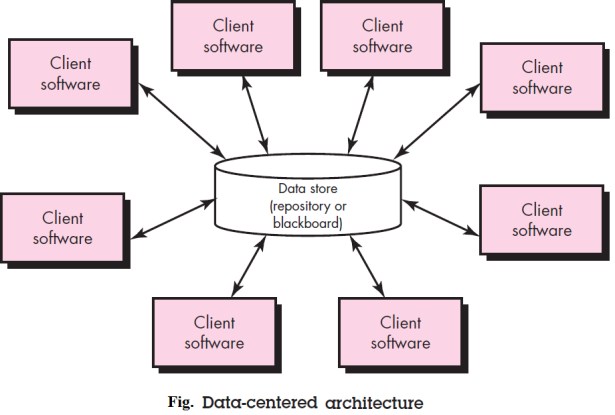
The architectural design for software is the equivalent to the floor plan of a house that give us the overall view of the building. Similarly, the software architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

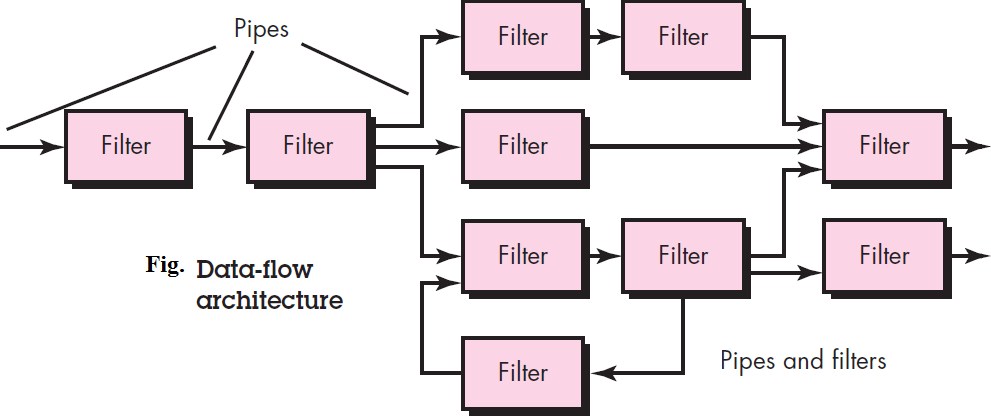
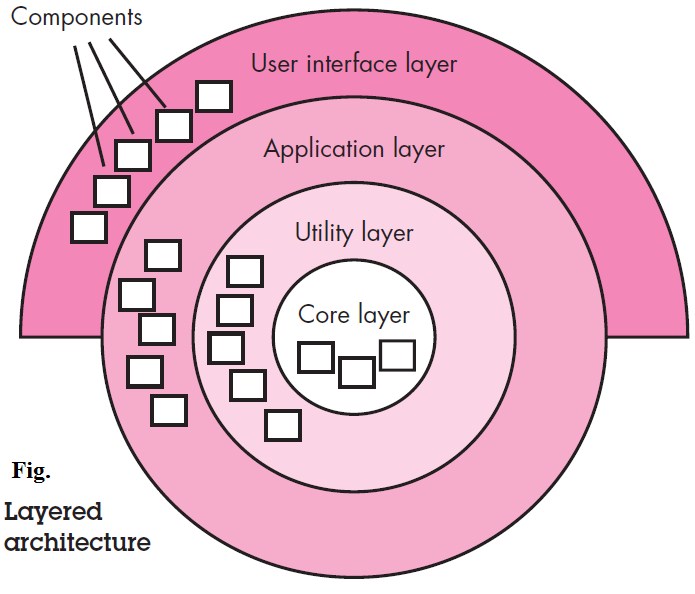
Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method. The architectural model is derived from three sources:

* Information about the application domain for the software to be built;
* Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and
* The availability of architectural styles and patterns.

***Architectural Style***

As the various architecture of building construction, the software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

* A set of components (e.g., a database, computational modules) that perform a function required by a system;
* A set of connectors that enable “communication, coordination and cooperation” among components;
* Constraints that define how components can be integrated to form the system; and
* Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

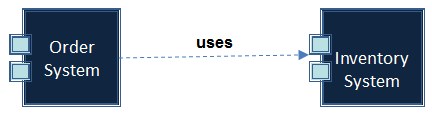
1. **Data-centered architectures:** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure illustrates a typical data-centered style.
2. **Data-flow architectures:** 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 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.
3. **Call and return architectures**. This architectural style enables you 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 illustrates architecture of this type.
   * **Remote procedure calls architectures**. The components of main program/subprogram architecture are distributed across multiple computers on a network.
4. **Object-oriented architectures:** The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.
5. **Layered architectures:** 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.

**Component Level Design**

A component is a modular building block for computer software that encapsulates implementation and exposes a set of interfaces. The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

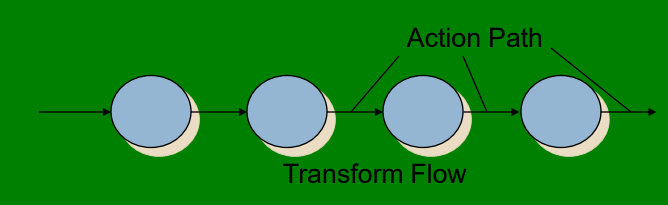
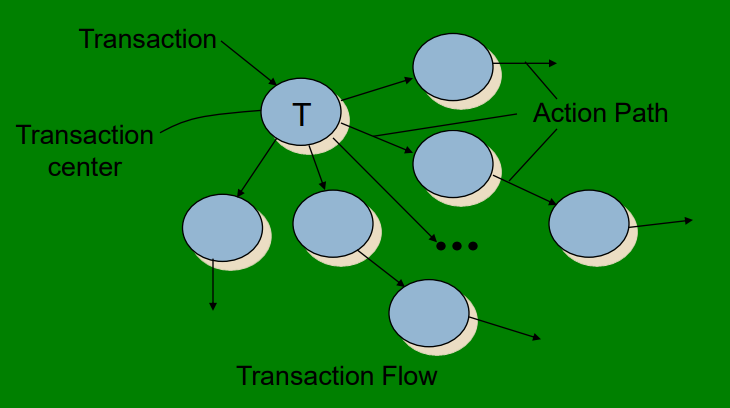
* **Component Diagram**

The component diagram is used to model the physical aspects of an OO system. It shows the software components of a system & how they are related to each other. Thus, they show the organization and dependencies between a set of components. Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations. Use component diagrams to model the static implementation view of a system. This involves modeling the physical things that reside on a node, such as executable, libraries, tables, files, and documents.

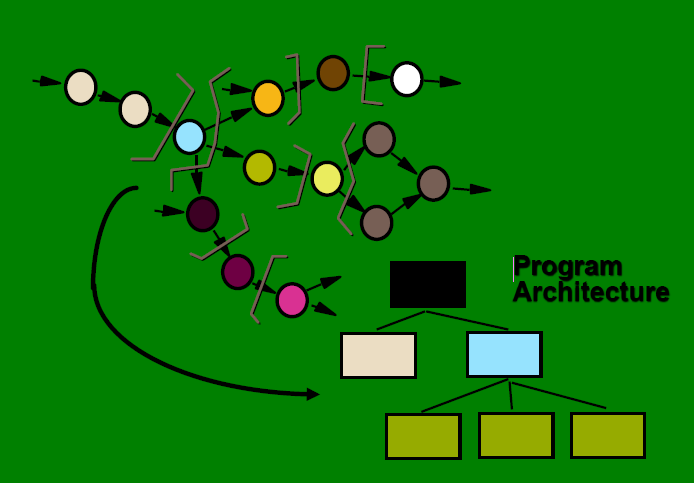
**Mapping Requirements to Software Architecture**

Establish type of information flow

* Transform flow -overall data flow is sequential and flows along a small number of straight line paths
* Transaction flow - a single data item triggers information flow along one of many paths.

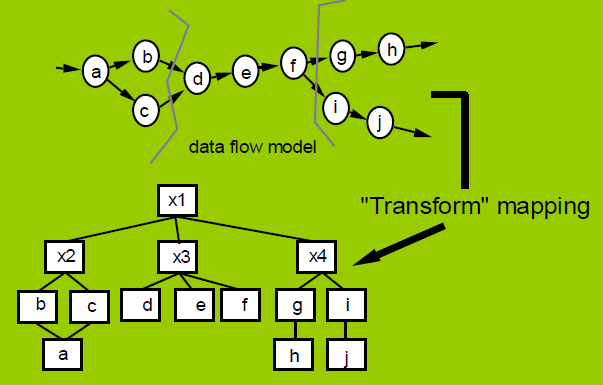


* Flow boundaries indicated
* DFD is mapped into program structure
* Control hierarchy defined
* Resultant structure refined using design measures and heuristics
* Architectural description refined and elaborated

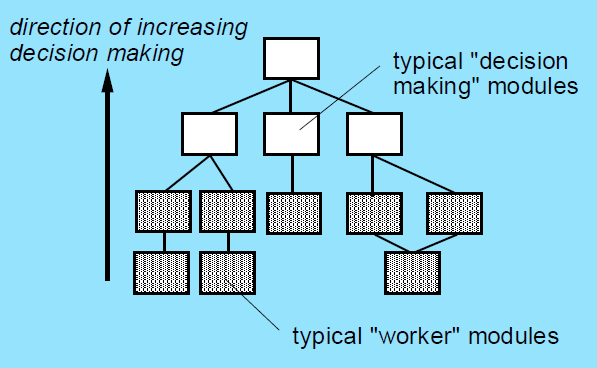


**Transform Mapping**

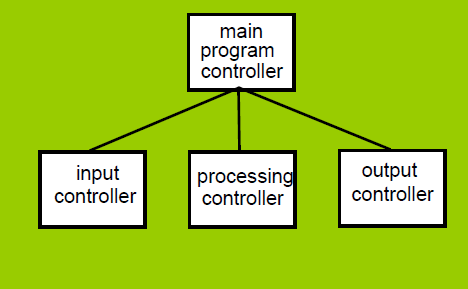
* Review fundamental system model
* Review and refine data flow diagrams for the software
* Determine whether the DFD has transform or transaction characteristics
* Isolate the transform center by specifying incoming and outgoing flow boundaries
* Perform first level factoring
* Perform second level factoring
* Refine the first iteration architecture using design heuristics for improved software quality



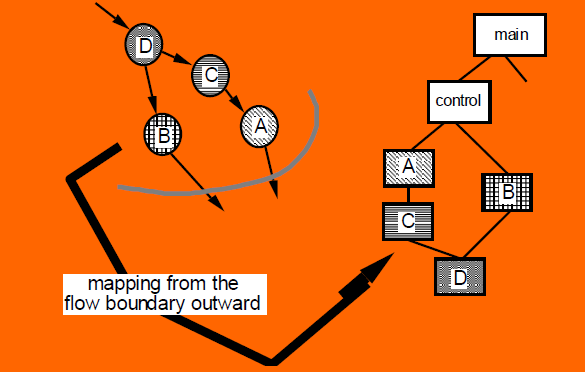
**Factoring**



**First Level Factoring**

****

**Second level factoring**

****