BCA V SE
Unit - III

Software Design - Objectives

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.

Objectives

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes —good‖ software, and an overall commitment to quality are critical success factors for a competent design. The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process. Davis [DAV95] suggests a set of principles for software design, which have been adapted and extended in the following list:

- **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, 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, inconsistency) have been addressed before worrying about the syntax of the design model.

Design Concepts

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are as follows:

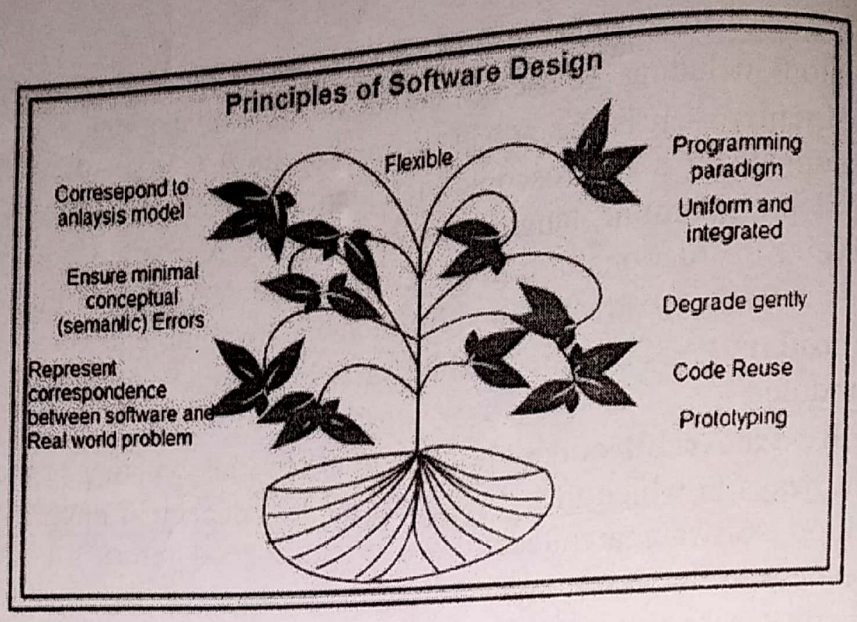
1. **Abstraction -** Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant

for a particular purpose.It is an act of Representing essential features without including the background details or explanations.

2. **Refinement -** It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.
3. **Modularity -** Software architecture is divided into components called modules.
4. **Software Architecture -** It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. Good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule and cost.
5. **Control Hierarchy -** A program structure that represents the organization of a program component and implies a hierarchy of control.
6. **Structural Partitioning -** The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.
7. **Data Structure -** It is a representation of the logical relationship among individual elements of data.
8. **Software Procedure -** It focuses on the processing of each module individually.
9. **Information Hiding -** Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice.



Some of the commonly followed design principles are as following.

1. **Software design should correspond to the analysis model:** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.
2. **Choose the right programming paradigm:** A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.
3. **Software design should be uniform and integrated:** Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.
4. **Software design should be flexible:** Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.
5. **Software design should ensure minimal conceptual (semantic) errors:** The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.
6. **Software design should be structured to degrade gently:** Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.
7. **Software design should represent correspondence between the software and real-world problem:** The software design should be structured in such away that it always relates with the real-world problem.
8. **Software reuse:** Software engineers believe on the phrase: 'do not reinvent the wheel'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.
9. **Designing for testability:** A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed (designed and implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer.

However, it has become apparent that the process of separating testing is seriously flawed, as if any type of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.

10. **Prototyping:** Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick 'mock-up' of the system can be developed. This mock-up can be used as a effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

Note that design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures, and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references and maintenance.

The Software Design Methodology

Software design is an important activity as it determines how the whole software development task would proceed including the system maintenance. The design of software is essentially a skill, but it usually requires a structure which will provide a guide or a methodology for this task. A methodology can be defined as the underlying principles and rules that govern a system. A method can be defined as a systematic procedure for a set of activities. Thus, from these definitions, a methodology will encompass the methods used within the methodology. Different methodologies can support work in different phases of the system life cycle

Data Design in Software Engineering

Data design is the first design activity, which results in fewer complexes, 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.

1. The data structures needed for implementing the software as well-as the operations that can be applied on them should be identified.
2. 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.
3. Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
4. Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
5. A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.

6. 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.

The Design Process

Design is a formation of a plan of activities to accomplish a recognized need. The need may be well defined. When needs are well-defined, it is likely due to the fact that neither the need nor problem has been identified. The design process is a process of creative invention and definition, it involves synthesis and analysis, and thus, is difficult to summarize in a simple design formula.

Design is an applied science. In a software design problem, a number of solutions exist. The designer (each word "designer" can also refer to "designers") must plan and execute the design strategy taking into account certain established design practices.

The designer often has to fall back on previous experience gained and has to study the existing software methodologies designed by others, to analyze their advantages and disadvantages. It is useful also to review the basic parameters, especially the requirements and system specifications. A designer must constantly improve and enrich his store of design solution. The development of design alternatives should be a regular design activity aimed at seeing the most rational solution. In the design of software, often there are different design methodologies that can be used to derive a software solution, this is called the design degree of freedom.

A design solution can also have several degrees of freedom, which implies that there is possibly at least one solution in the solution space, often there is more than one solution. It must be noted that there are no unique answers for design. However, that does not imply that any answer will do. Some solutions are more optimal than others. A design is subject to certain problem-solving constraints, for example, in a vacation design problem, the constraints are money and time. Note also that there are constraints on the solutions, for example, users must be satisfied. Then in synthesis, it is necessary to begin with the solution of the main design problems and separate secondary items from the main ones. Successful design often begins with a clear definition of the design objectives. While in analysis and evaluation, the designer using these knowledge and simple diagrams, makes a first draft of the design in the form of diagrams. This has to be thoroughly analyzed to verify that it meets the design objectives and that it is adequate but not over designed. A systematic approach is useful only to the extent that the designer is presented with a strategy that he can use as a base for planning the required design strategy for the problem at hand. A design problem is not a theoretical construct. Design has an authentic purpose - the creation of an end result by taking definite action or the creation of something having physical reality. The design process is by necessity an iterative one. The software design should describe a system that meets the requirements.

"The process of design consists of taking a specification (a description of what is required) and converting it into a design (a description of what is to be built). A good design is complete (it will

build everything required), economical (it will not build what is not required), constructive (it says how to build the product), uniform (it uses the same building techniques throughout), and testable (it can be shown to work)."

Most software engineers agreed that software design integrates and utilizes a great deal of the basics of computer science and information systems; unless one can apply the knowledge gained to the design of quality software, one cannot truly be considered to have mastered software design.

The Role of Design Methodology

The role of the software design methodology cannot be overemphasized (Freeman, 1980). Software design methodology provides a logical and systematic means of proceeding with the design process as well as a set of guidelines for decision-making. The desi gn methodology provides a sequence of activities, and often uses a set of notations or diagrams. The design methodology is especially important for large complex projects that involve programming-in-the-large (where many designers are involved); the use of a methodology establishes a set of common communication channels for translating design to codes and a set of common objectives. In addition, there must be an objective match between the overall character of the problem and the features of the solutio n approach, in order for a methodology to be efficient.

Design Phase in Software Systems Development

Pressman (1992) has described the software development process as consisting of three broad generic phases - the definition, development and maintenance phases. The definition phase defines the "what" of the software system, the development phase define s the "how" and the maintenance phase defines the support and future necessary changes. Accordingly, software design is placed in the development phase of Pressman's Software Life Cycle model. There are other models of the Software Development Life Cyc le (SDLC) model (see Fairley, 1985; Burch, 1992; Sage & Palmer, 1990). Almost every text on software development includes a SDLC model, there are some variations but, in general, the basic phases or activities are always present. The basic phases that are ever present are the analysis, design, testing, implementation and maintenance phases.

The analysis phase involves the requirement definition, from which the software specifications are derived. Design then translates the specification into a set of notations that represents the algorithms, data structures, architecture and interfaces. The representations are then coded and tested for defects in function, logic and implementation. When the software is ready, it is implemented and maintained by support personnel.

While each of the analysis, design, testing, implementation and maintenance phases need to be performed in all cases, there are a number of ways their interactions can be organized. The major models that are in use are the Waterfall model (Royce, 1970; Boehm, 1976), Prototyping Model (McCracken & Jackson, 1982; Gladden 1982) and the Spiral model (Boehm, 1986).

In all the different models, design plays a central role in the models. Software design is thus a major phase in the software system development. In this research project, the design process of the SDLC will be considered, which includes requirement def inition or system analysis, system or requirement specifications, logical or system design, and detailed or program design and development. Even though pure software design consists of architectural and detailed design. Pure software design cannot proce ed without the requirement definition and specification stages. Architectural design deals with the general structure of a software system. It involves identifying and decomposing the software into smaller models or components, determining data structur es and specifying the relationship among the modules. According to a Software Engineering Institute report (Budgen, 1989, p6), detailed design involves the "formulation of blueprints for the particular solution and with modeling the detailed interaction between its components." It is concerned with the detailed "how to" for packaging all the components together.

The classification of software design approaches

Level-Oriented Design

In the level-oriented design approach, there are two general or broad strategies that can be used. The first strategy starts with a general definition of a solution to the problem then through a step-by-step process produce a detailed solution (this is called Stepwise Refinement). This is basically dependent on the system requirements and is a top-down process. The other strategy is to start with a basic solution to the problem and through a process of modeling the problem, build up or extend the solution by adding additional features (this is called design by composition).

The top-down process starts at the top level and by functional decomposition, breaks down the system into smaller functional modules. Smaller modules are more readily analyzed, easier to design and code. But, inherent in the top-down process is the req uirement that there must be a complete understanding of the problem or system at hand. Otherwise, it could lead to extensive redesign later on. The top-down process also is dependent on decisions made at the early stages to determine the design structure re. Different decisions made at the early stage will result in different design structures. Functional decomposition is an iterative "break down" process called stepwise refinement, where each level is decomposed to a more detailed lower level.

Thus, at each decomposition, there have to be a way to determine if further decomposition is needed or necessary, that is, if the atomic level has been achieved. There are no inherent procedure or guidelines for this. There is also a possibility of duplication if stepwise refinement is not done carefully or "correctly"; this will occur toward the end of the process, that is, at the lower levels. This can be costly, especially if there are many different designers or programming teams working on a single system. As a result, the top-down process is often used in the initial phase of the design process to break down the different components or modules of a system. The top-down process has also been used as a preliminary step in the other design methodologies. Once the modules of the system have been determined, they can be divided amongst the different designers or design teams.

The design by composition strategy involves the evolution of a solution by building upon the solution from the previous stage. Using this technique, additional features are added as the solution evolves. This strategy uses as its origin, the basic or simple initial solution and through an iterative composition process add or expand the solution to include additional modules. This approach will also encompass the bottom-up design, where the lowest level solution is developed first and gradually builds up to the highest level. Freeman (1983) has added a few models, such as the outside-in model where what the end-users sees (external functions of the system) are defined as the top-level decisions and the implementation (the inside of the system) as the lower-level decisions. This model was created to overcome the tendency of designers to pay insufficient attention to the needs of end-users. The alternative to the outside-in model is the inside-out model, where decisions relating to the implementation (inside) of the system are made before the external function of the system. Another model is based on the most-critical-component-first approach, where one first design the components of the systems that are the most constrained so that these critical parameters are satisfied. Then the rest of the system components are designed. Often these models are conceptual, not to be rigorously enforced because in a real design effort, integration of models is often necessary.

Data Flow-Oriented Design

In the data flow-oriented design approach, which is often called Structured Design, information flow characteristic is used to derived program structure. In the data flow-oriented approach, emphasis is on the processing or operations performed on the da ta. Design is information driven. Information may be represented as a continuous flow that is transformed, as it is processed from node to node in the input-output stream. As software can ideally be represented by a data flow diagram (DFD), a design mod el that uses a DFD can theoretically be applied in the software development project. The data flow-oriented approach is especially applicable when information is processed

without hierarchical structure. A DFD can be mapped into the design structure by two means - transform analysis or transaction analysis. Transform analysis is applied when the data flow in the input-output stream has clear boundaries. The DFD is mapped into a structure that allocates control to three basic modules - input, process and output. Transaction analysis is applied when a single information item causes flow to branch along one of many paths. The DFD is mapped to a substructure that acquires and evaluates a transaction; another substructure controls all the data processing actions based on a transaction. A few examples of structured design or data flow-oriented design methodologies are Structured Analysis and Design Technique (SADT), Systematic Activity Modeling Method (SAMM) and Structured Design (SD).

Data Structure-Oriented Design

The data structure-oriented design approach utilizes the data structures of the input data, internal data (for example databases) and output data to develop software. In the data structure-oriented approach, the emphasis is on the object, which is the data. The structure of information, called data structure, has an important impact on the complexity and efficiency of algorithms designed to process information.

Software design is closely related to the data structure of the system, for example, alternative data will require a conditional processing element, repetitive data will require a control feature for repetition and a hierarchical data structure will require a hierarchical software structure. Data structure-oriented design is best utilized in applications that are a well-defined, hierarchical structure of information.

As both data flow and data structure oriented design approaches are based on considerations in the information domain, there are similarities between both approaches. Both depend on the analysis step to build the foundation for later steps. Both attempt to transform information into a software structure; both are driven by information. In data structure-oriented design information structure are represented using hierarchical diagrams; DFD has little relevance; transformation and transaction flows are not considered. Data structure-oriented design have a few tasks - evaluate the characteristics of the data structure, represent the data in its lowest form such as repetition, sequence or selection, map the data representation into a control hierarchy for software, refine the control hierarchy and then develop a procedural description of the software. Some examples of the data structure-oriented design approach are the Jackson System Development (JSD) and the Data Structured Systems Development (DDSD) which is also called the Warnier-Orr methodology.

Object Oriented Design

The object oriented design approach is unique in its usage of the three software design concepts: abstraction, information hiding and modularity. Objects are basically a producer or consumers of information or an information item. The object consists of a private data structure and related operations that may transform the data structure. Operations contain procedural and control constructs that may be invoked by a message, that is, a request to the object to perform one of its operations. The object also has an interface where messages are passed to specify what operation on the object is desired. The object that receives a message will then determine how the requested operation is to be performed. By this means, information hiding (that is, the details of implementation are hidden form all the elements outside the object) is achieved. Also objects and their operations are inherently modular, that is, software elements (data and process) are grouped together with a well-defined interface mechanism (that is, messages).

Object oriented design is based on the concepts of: objects and attributes, classes and members, wholes and parts. All objects encapsulate data (the attribute values that define the data), other objects (composite objects can be defined), constants (set values), and other related information. Encapsulation means that all of this information is packaged into a single name and can be re-used. The object oriented design is rather new and as such it is still evolving even at this present moment. Object oriented design encompasses data design, architectural design and procedural design. By identifying classes and objects, data abstractions are created; by coupling operations to data, modules are specified and a structure for the software is established; by developing a mechanism for using objects (for example, passing of messages) interfaces are described.

Object-oriented Architecture comes in above design concept

In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. In this style, components are represented as *objects* and they interact with each other through methods (connectors). This architectural style has two important characteristics, which are listed below.

1. Objects maintain the integrity of the system.
2. An object is not aware of the representation of other objects.
3. Some of the advantages associated with the object-oriented architecture are listed below.
4. It allows designers to decompose a problem into a collection of independent objects.
5. The implementation detail of objects is hidden from each other and hence, they can be changed without affecting other objects.

Architectural Design in Software Engineering

Requirements of the software should be transformed into an architecture that describes the software's top-level structure and identifies its components. This is accomplished through architectural design (also called **system design),** which acts as a preliminary 'blueprint' from which software can be developed. **IEEE** defines architectural design as 'the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.' This framework is established by examining the software requirements document and designing a model for providing implementation details. These details are used to specify the components of the system along with their inputs, outputs, functions, and the interaction between them. An architectural design performs the following functions.

1. It defines an abstraction level at which the designers can specify the functional and performance behaviour of the system.
2. It acts as a guideline for enhancing the system (when ever required) by describing those features of the system that can be modified easily without affecting the system integrity.
3. It evaluates all top-level designs.
4. It develops and documents top-level design for the external and internal interfaces.
5. It develops preliminary versions of user documentation.
6. It defines and documents preliminary test requirements and the schedule for software integration.
7. The sources of architectural design are listed below.
8. Information regarding the application domain for the software to be developed
9. Using data-flow diagrams
10. Availability of architectural patterns and architectural styles.

Architectural design is of crucial importance in software engineering during which the essential requirements like reliability, cost, and performance are dealt with. This task is cumbersome as the software engineering paradigm is shifting from monolithic, stand-alone, built-from-scratch systems to componentized, evolvable, standards-based, and product line-oriented systems. Also, a key challenge for designers is to know precisely how to proceed from requirements to architectural design. To avoid these problems, designers adopt strategies such as reusability, componentization, platform-based, standards-based, and so on.

Though the architectural design is the responsibility of developers, some other people like user representatives, systems engineers, hardware engineers, and operations

personnel are also involved. All these stakeholders must also be consulted while reviewing the architectural design in order to minimize the risks and errors.

Architectural Design Representation

Architectural design can be represented using the following models.

1. **Structural model:** Illustrates architecture as an ordered collection of program components
2. **Dynamic model:** Specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment
3. **Process model:** Focuses on the design of the business or technical process, which must be implemented in the system
4. **Functional model:** Represents the functional hierarchy of a system
5. **Framework model:** Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

Architectural Design Output

The architectural design process results in an **Architectural Design Document (ADD).** This document consists of a number of graphical representations thatcomprises software models along with associated descriptive text. The softwaremodels include static model, interface model, relationship model, and dynamic processmodel. They show how the system is organized into a process at run-time.

Architectural design document gives the developers a solution to the problem stated in the Software Requirements Specification (SRS). Note that it considers only those requirements in detail that affect the program structure. In addition to ADD, other outputs of the architectural design are listed below.

1. Various reports including audit report, progress report, and configuration status accounts report
2. Various plans for detailed design phase, which include the following
3. Software verification and validation plan
4. Software configuration management plan
5. Software quality assurance plan
6. Software project management plan.

Architectural Styles

Architectural styles define a group of interlinked systems that share structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system. If an existing architecture is to be re-engineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes re-assignment of the functionality performed by the components.

By applying certain constraints on the design space, we can make different style-specific analysis from an architectural style. In addition, if conventional structures are used for an architectural style, the other stakeholders can easily understand the organization of the system.

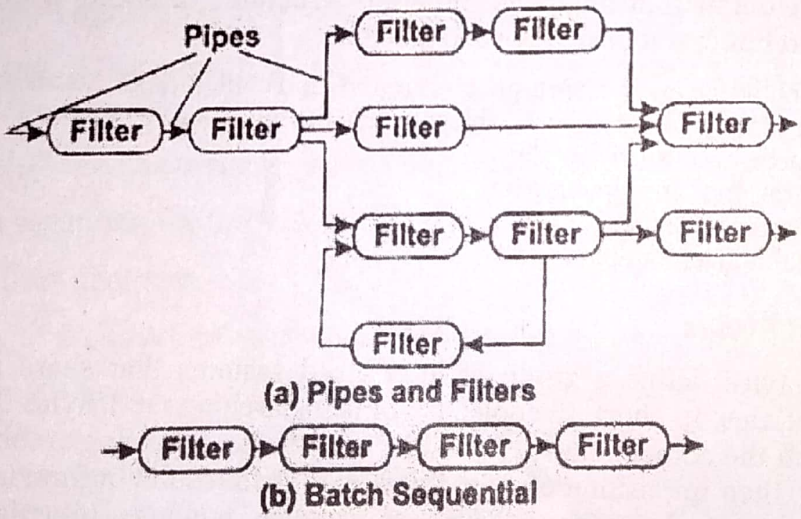
A computer-based system (software is part of this system) exhibits one of the many available architectural styles. Every architectural style describes a system category that includes the following.

1. Computational components such as clients, server, filter, and database to execute the desired system function
2. A set of *connectors* such as procedure call, events broadcast, database protocols, and pipes to provide communication among the computational components
3. Constraints to define integration of components to form a system
4. A *semantic* model, which enable the software designer to identify the characteristics of the system as a whole by studying the characteristics of its components.

Some of the commonly used architectural styles are data-flow architecture, object oriented architecture, layered system architecture, data-centered architecture, and call and return architecture. Note that the use of an appropriate architectural style promotes design reuse, leads to code reuse, and supports interoperability.

Data-flow Architecture

Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations. Each component, known as **filter,** transforms the data and sends this transformed data to other filters for further processing using the connector, known as **pipe**. Each filter works as an independent entity, that is, it is not concerned with the filter which is producing or consuming the data. A pipe is a unidirectional channel which transports the data received on one end to the other end. It does not change the data in anyway; it merely supplies the data to the filter on the receiver end.



Data-flow Architecture

Most of the times, the data-flow architecture degenerates a batch sequential system. In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data. One common example of this architecture is UNIX shell programs. In these programs, UNIX processes act as filters and the file system through which UNIX processes interact, act as pipes. Other well-known examples of this

architecture are compilers, signal processing systems, parallel programming, functional programming, and distributed systems. Some **advantages** associated with the data-flow architecture are listed below.

1. It supports reusability.
2. It is maintainable and modifiable.
3. It supports concurrent execution.
4. Some disadvantages associated with the data-flow architecture are listed below.
5. It often degenerates to batch sequential system.
6. It does not provide enough support for applications requires user interaction.
7. It is difficult to synchronize two different but related streams.

Procedural Design

The procedural design is often understood as a software design process that uses mainly control commands such as: sequence, condition, repetition, which are applied to the predefined data. Sequences serve to achieve the processing steps in order that is essential in the specification of any algorithm.

The procedural design is often understood as a software design process that uses mainly control commands such as: sequence, condition, repetition, which are applied to the predefined data.

Sequences serve to achieve the processing steps in order that is essential in the specification of any algorithm.

Conditions provide facilities for achieving selected processing according to some logical statement.

Repetitions serve to achieve loopings during the computation process.

These three commands are implemented as ready programming language constructs.

The programming languages that provide such command constructs are called imperative programming languages.

The software design technique that relies on these constructs is called procedural design, or also structured design.