Component design introduction

- A software component is a modular building block for the computer software.
- Component is defined as a modular, deployable and replaceable part of the system which encloses the implementation and exposes a set of interfaces.

Components view

The components has different views as follows:

1. An object-oriented view

- An object-oriented view is a set of collaborating classes.
- The class inside a component is completely elaborated and it consists of all the attributes and operations which are applicable to its implementation.
- To achieve object-oriented design it elaborates analysis classes and the infrastructure classes(design).

2. The traditional view/Conventional View

- A traditional component is known as module.
- It is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it.
- It resides in the software and serves three important roles which are control component, a problem domain component and an infrastructure component.
- A control component coordinate is an invocation of all other problem domain components.
- A problem domain component implements a complete function which is needed by the customer.
- An infrastructure component is responsible for function which support the processing needed in the problem domain.

3. The Process related view

- This view highlights the building system out of existing components.

- The design patterns are selected from a catalog and used to populate the architecture. In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library. As the software architecture is formulated, components are selected from the library and used to populate the architecture.

Class-based design components

The principles for class-based design component are as follows:

1. Open Closed Principle (OCP)
Any module in OCP should be available for extension and modification.

–A module or component should be open for extension but closed for modification

–The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component

2. The Liskov Substitution Principle (LSP)

- The subclass must be substitutable (capable of being exchanged for another) for their base class.
- A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- This principle was suggested by Liskov.

3. **Dependency Inversion Principle (DIP)**

- It depends on the abstraction and not on concretion (A concretion is a real world instance of something that implements the functionality described by the abstraction. When we are writing code, we call the abstractions Interfaces and the concretions Implementations).
- Abstraction is the place where the design is extended without difficulty.

4. The Interface Segregation Principle (ISP)
Many client specific interfaces is better than the general purpose interface. There are many instances in which multiple client components uses the operations provided by server class.

5. The Release Reuse Equivalency Principle (REP)

- A fragment of reuse is the fragment of release.
- The class components are designed for reuse which is an indirect contract between the developer and the user.

6. The common closure principle (CCP)
The classes change and belong together i.e the classes are packaged as part of design which should have the same address and functional area. For Example: when some characterstics of that area is changed , it is likely that only those classes within that package require modification.

7. The Common Reuse Principle (CRP)
The classes that are not reused together should not be grouped together.

Classes that are reused together should be included within a same package.

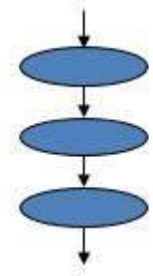
Traditional design components

- Introduction : Traditional components are designed based on different constructs like
  - Sequence,
  - Condition,
  - Repetition.
- **Sequence** implements processing steps that are essential in the specification of any algorithm.
- **Condition** provides the facility for selected processing based on some logical occurrence.
- **Repetition** allows for looping.
- **These three constructs are fundamental to structured programming— an important component-level design technique.**
- The use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability.

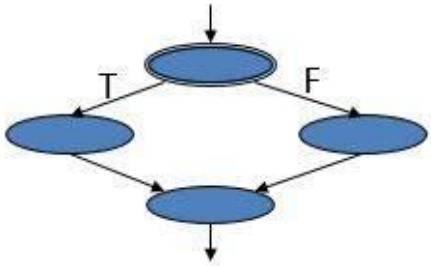
- **Graphical Design Notation**
  - A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words.

- There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail.
- The activity diagram allows you to represent sequence, condition, and repetition— all elements of structured programming.
- It is a descendent of an earlier pictorial design representation (still used widely) called a flowchart.
- A flowchart, like an activity diagram, is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control

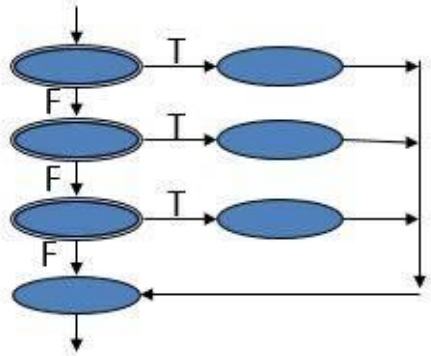
- Graphical Design Notation



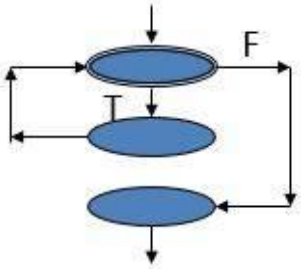
Sequence



If-then-else



Selection



Repetition

-

- **Tabular Design Notation**
  - In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions.

- Decision tables provide a notation that translates actions and conditions into a tabular form.
- The table is difficult to misinterpret and may even be used as a machine-readable input to a table-driven algorithm.

Tabular Design Notation

1)List all actions that can be associated with a specific procedure (or module)

2)List all conditions (or decisions made) during execution of the procedure

3)Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions

4)Define rules by indicating what action(s) occurs for a set of conditions

	Rules			
Conditions	1	2	3	4
Condition A	T	T		F
Condition B		F	T	
Condition C	T			T
Actions				
Action X	**P**		**P**	
Action Y				**P**
Action Z	**P**	**P**		**P**