COMPONENT-BASED ARCHITECTURE

Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based architecture is to ensure **component reusability**. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit. There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services. These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation.

Component-oriented software design has many advantages over the traditional object-oriented approaches such as −

- Reduced time in market and the development cost by reusing existing components.
- Increased reliability with the reuse of the existing components.

What is a Component?

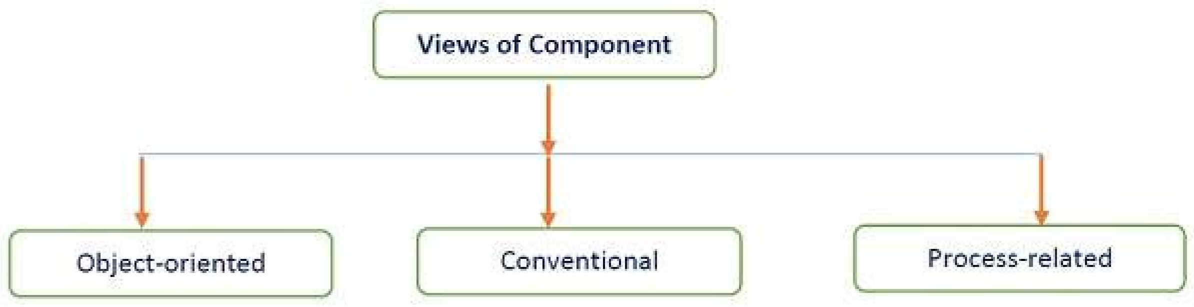
A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.

A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities. It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture.

A software component can be defined as a unit of composition with a contractually specified interface and explicit context dependencies only. That is, a software component can be deployed independently and is subject to composition by third parties.

Views of a Component

A component can have three different views − object-oriented view, conventional view, and process-related view.



Object-oriented view

A component is viewed as a set of one or more cooperating classes. Each problem domain class *analysis* and infrastructure class *design* are explained to identify all attributes and operations that apply to its implementation. It also involves defining the interfaces that enable classes to communicate and cooperate.

Conventional view

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.

Process-related view

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.

- A user interface *UI* component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.
- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time *JIT* approach.
- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean *EJB*, .NET components, and CORBA components.

Characteristics of Components

- **Reusability** − Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task.
- **Replaceable** − Components may be freely substituted with other similar components.
- **Not context specific** − Components are designed to operate in different environments and contexts.
- **Extensible** − A component can be extended from existing components to provide new behavior.
- **Encapsulated** − A A component depicts the interfaces, which allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state.
- **Independent** − Components are designed to have minimal dependencies on other components.

Principles of Component−Based Design

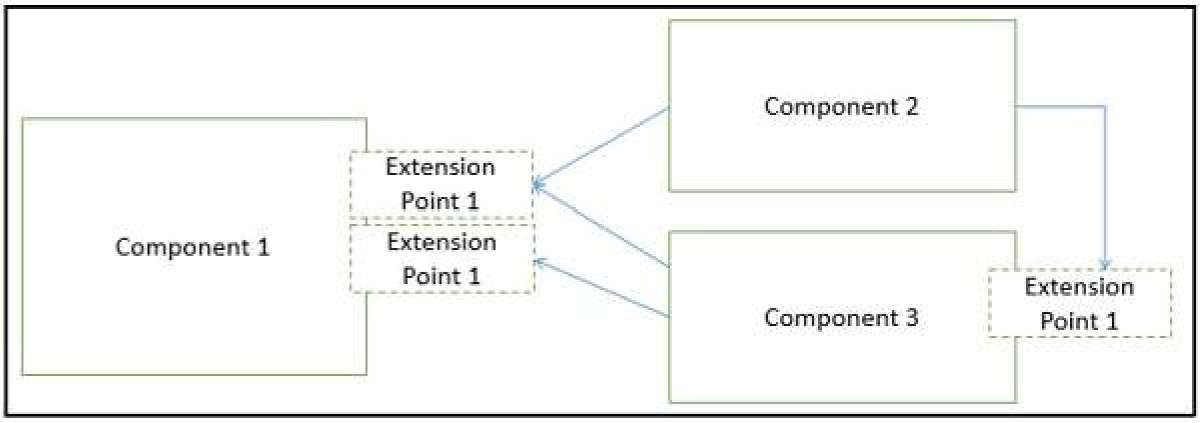
A component-level design can be represented by using some intermediary representation *e. g. graphical, tabular, ortext − based* that can be translated into source code. The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors.

It has following salient features −

- The software system is decomposed into reusable, cohesive, and encapsulated component units.
- Each component has its own interface that specifies required ports and provided ports; each component hides its detailed implementation.
- A component should be extended without the need to make internal code or design modifications to the existing parts of the component.
- Depend on abstractions component do not depend on other concrete components, which increase difficulty in expendability.
- Connectors connected components, specifying and ruling the interaction among components. The interaction type is specified by the interfaces of the components.
- Components interaction can take the form of method invocations, asynchronous invocations, broadcasting, message driven interactions, data stream communications, and other protocol specific interactions.
- For a server class, specialized interfaces should be created to serve major categories of

clients. Only those operations that are relevant to a particular category of clients should be specified in the interface.

- A component can extend to other components and still offer its own extension points. It is the concept of plug-in based architecture. This allows a plugin to offer another plugin API.



Component-Level Design Guidelines

It creates a naming conventions for components that are specified as part of the architectural model and then refines or elaborates as part of the component-level model.

- Attains architectural component names from the problem domain and ensures that they have meaning to all stakeholders who view the architectural model.
- Extracts the business process entities that can exist independently without any associated dependency on other entities.
- Recognizes and discover these independent entities as new components.
- Uses infrastructure component names that reflect their implementation-specific meaning.
- Models any dependencies from left to right and inheritance from top *baseclass* to bottom *derivedclasses*.
- Model any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

Conducting Component-Level Design

It recognizes all design classes that correspond to the problem domain as defined in the analysis model and architectural model.

This design has following important features −

- Recognizes all design classes that correspond to the infrastructure domain.
- Describes all design classes that are not acquired as reusable components, and specifies message details.
- Identifies appropriate interfaces for each component and elaborates attributes and defines data types and data structures required to implement them.
- Describes processing flow within each operation in detail by means of pseudo code or UML activity diagrams.
- Describes persistent data sources *databasesandfiles* and identifies the classes required to manage them.
- Develops and elaborates behavioral representations for a class or component. This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class.
- Elaborates deployment diagrams to provide additional implementation detail.

- Demonstrates the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environment.
- The final decision can be made by using established design principles and guidelines. Experienced designers consider all *ormost* of the alternative design solutions before settling on the final design model.

Advantages

- **Ease of deployment** − As new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole.
- **Reduced cost** − The use of third-party components allows you to spread the cost of development and maintenance.
- **Ease of development** − Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- **Reusable** − The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- **Modification of technical complexity** − A component modifies the complexity through the use of a component container and its services.
- **Reliability** − The overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse.
- **System maintenance and evolution** − Easy to change and update the implementation without affecting the rest of the system.
- **Independent** − Independency and flexible connectivity of components. Independent development of components by different group in parallel. Productivity for the software development and future software development.

Loading [MathJax]/jax/output/HTML-CSS/jax.js