Chapter 4: Components

Software architecting is the act of creating a representation of an unknown and original object whose properties must be well enough understood in advance. Software architecture is the work product emanating out of architecting and expressed using architecture descriptions. It is the structure of components of a system, their relationships, externally visible properties, and principles and guidelines governing design and evolution over time [26-27]. This thesis considers components as a notional environment that lies within the confines of software, and represents the most fundamental element of software that enables it to perform its function. Accordingly, components are the building blocks for software architecture. Initially, a component is a platonic thing, an idea, a type. It is manifest as an independent entity when it is instantiated and becomes a resource. Composition of such resources leads to software architecture. Components are defined through descriptions in architecture description languages. This would be the direction taken in this chapter in order to illustrate the nature of software Plug and Play. Going further, the first step is to arrive at a definition of component that clarifies the author’s position on what a component is. Before that, a literature survey of the existing definitions of components is taken up so that the proposed definition can be seen in the context of past work.

4.1 Review of Literature on definition of components

According to Allen, Garlan, Shaw and Oquendo [43-44][55-56][86], a component describes a localized, independent computation that corresponds to compilation units of programming languages and other user-level objects. In other words, Components are the locus of computation and state. Each Component is further defined by a component type description comprising of an interface (which describes its properties) and a computation. While the interface consists of ports which represent an interaction in which the component may participate, the computation describes what the component actually does, (it carries out the interactions described by the ports and is a full specification upon which the components properties are based). It is evident that Allen, Garlan, Shaw and Oquendo look at Components simply as those Software elements that perform computation. In other words, they treat Components predominantly as computational units.

According to Bruneton et al [84-85], in the FRACTAL Component model, a component is a run-time entity that is encapsulated and has a distinct identity. Further, at the lowest level of control, a FRACTAL Component is a black box that does not provide any introspection or intercession capability. A FRACTAL Component provides an interface that allows one to discover all its external interfaces. An interface is an
access point to a component and supports a finite set of operations. Each interface has a unique identity that distinguishes it from the others. At the highest level of control, a FRACTAL Component is composed of a membrane which provides external interfaces to introspect and reconfigure its internal features and a content that consists in a finite set of other sub-Components. The membrane of a FRACTAL Component has external interfaces that are accessible from outside the component and internal interfaces that are accessible only for its sub-Components. From this, it is evident that Bruneton et al, also look at Components as computational entities.

According to Fielding [87], Components are the most easily recognized element of a software architecture. Fielding defines Components as an abstract unit of software instructions and internal state that provides a transformation of data via its interface. Further, the behaviour of each Component is part of the architecture in so far as that behaviour can be observed or discerned from the point of view of another Component. In other words, Fielding suggests that a component is defined by its interface and the service it provides to other components, rather than by its implementation behind the interface. It is evident that Fielding has an interface oriented view of components and his target Architecture is the interface connection Architecture.

According to Szyperski [88], a component encapsulates its constituent features and is self-contained. It provides specification of what it requires and what it provides (in terms of services). It encapsulates its implementation and interacts with its environment through well-defined interfaces. The characteristic properties of a component are that they are a unit of independent deployment, they are a unit of third-party Composition and they have no externally observable state. It is evident that Szyperski views Components as executable units (reusable assets) of independent production, acquisition and deployment that can be composed into a functioning System.

According to Achermann et al [83], Components are black-box entities that encapsulate services behind well-defined interfaces. Component provides and requires services (which can be seen as plugs/interfaces). A component that is not plug-compatible with any other another is not a component. The plugs of a component can be of different form, provide different functions and support different data schemas. By definition, Components are always elements of a component framework. They adhere to a particular Component Architecture or Architecture style that defines the plugs, the connectors to the components and the corresponding Composition rules. It is evident that Achermann et al view Components as Software elements that provide the required functionality to other Components in the form of Services.
According to Krutenhe [62], a component is a nontrivial, nearly independent, and replaceable part of a System that fulfils a clear function in the context of a well-defined Architecture. A component conforms to and provides the physical realization of a set of interfaces. Accordingly, a component is a reusable part of software, which is independently developed, and can be brought together with other Components to build larger units. It may be adapted but may not be modified by the System in which it is part of. A component has a set of interfaces that it provides to or requires from its environment. It is evident that Krutenhe views Components as a fundamental Software element that encapsulates information and interfaces.

According to Councill et al [81], a Software Component is a Software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. A component model defines specific interaction and Composition standards and its implementation is the dedicated set of executable Software elements required to support the execution of the components while conforming to the component model. Further, they are of the view that a component cannot be differentiated from other Software elements and its implementation is completely hidden and available only in binary form. It is evident that Councill et al look at Software Component as a sequence of abstract program statements that describe computations to be performed by a computing machine, in other words, Components are computational entities that conform to a component model.

According to Magee et al [89], in DARWIN Architecture description language, as well as Feiler et al, in the architecture Analysis and Design Language, Components are viewed in terms of both the services they provide to allow other Components to interact with them and the services they require to interact with other Components. According to them, a component is characterized by its identity (name and run-time essence), possible interfaces with other components, distinguishing properties (critical characteristics of a component within its architectural context), and sub-components and their interactions. Such Components may be implemented and tested independently of the rest of the System of which they form a part which permits the reuse of components during construction through multiple instantiation and simplifies replacement during maintenance. It is evident that Magee et al and Feiler et al, view Components in terms of the interfaces that it provides and consumes in order to offer services to other Components in the System.

According to Sifakis et al [71][73], the term ‘Component’ denotes any executable description whose runs can be modelled as a sequence of actions. Each Component can perform actions from a vocabulary
of actions. The behaviour of a component describes the effect of its actions. The various Component actions can be composed together to produce interactions. A System of interacting Components is a set of components integrated through various mechanisms for coordinating their execution. The overall effect of the integration is the restriction of the components behaviour within its environment. Components are further characterised by abstractions that ignore implementation details and describe properties relevant to their Composition. It is evident that Sifakis et al view Components as computational actions.

In UML [31], a component describes the physical part of a System and is a container of logical elements and represents things that participate in the execution of a System. Another definition of a component is that it is a structured class representing a modular part of a System with encapsulated content and whose manifestation is replaceable within its environment. A component defines its behaviours in terms of provided and required interfaces. It serves as a type whose conformance is defined by the provided and required interfaces. A component has an external view by means of interfaces thereby exposing its publicly visible properties and operations. It is evident that UML views Components as structural entities in the overall System schema.

According to Attiogbe et al [74], a component is a 8-tuple \( (W, \text{Init}, A, N, I, Ds, V, Cs) \) with:

\[
W = \{T, V, Vt, \text{Inv}\} \text{ is the state space, where } T \text{ is a set of types, } V \text{ is a set of Variables, } Vt \text{ is a set of typed variables and } \text{Inv} \text{ is the state invariant.}
\]

Init is the initialisation of the Vt variables declared as part of the state space.

A is a finite set of elementary actions.

N is a finite set of service names.

I is the component interfaces.

Ds is the set of service descriptions.

V is the function that maps service names to service descriptions.

Cs is a constraint related to the services of the interfaces in order to control their usage.

It is evident that the view taken by Attiogbe et al [50] is that components offers and provides services and the component schema supports primitives that facilitate this.

According to Lumpe [90], a Software Component is an element of a Software framework. A component framework is a collection of software Components with a software architecture that determines the interfaces that components may have and the rules governing their Composition. According to Lumpe, these definitions captures an essential characteristics of components, Components
are designed for Composition. Further, Lumpe states that a component is a Software abstraction that hides its implementation details and it must be instantiated to be used. The essential point in Lumpe’s definition is that components are not used in isolation but in the context of the software architecture, that determines how the components are plugged together. The framework and the rules of composition vary according to the purpose of the architecture. It is evident that the view of Lumpe with regard to the component is that components are created as part of a larger pool of components so that they can collectively achieve a purpose.

According to Lau et al [91-93], a software component is a Software unit with the following defining characteristics: a) encapsulation and b) compositionality. A component should encapsulate both data and computation. By data encapsulation, Lau et al mean that data should be private to the component. By computation encapsulation, Lau et al mean that computation happens entirely within itself. Another view of Lau et al is that components are constructed from computation units and exogenous connectors. In other words, an atomic Component is a pair (i, u), where, u is a computation unit and is an invocation connector that invokes u’s methods; i provides an interface to the component. A composite Component is a tuple (k, C1, C2, …Cj) for some j where k is a j-ary connector and each Ci is either an atomic Component or a composite Component. k is the composition connector that provides an interface to the component. It is evident that the view of Lau et al is Components encapsulate data and computation and are constructed from computation and connectors.

According to Sommerville [63], a component is a service provider of one or more services. His definition is an extension of the earlier definition that components are independent, fundamental unit of composition in a System. The view of component as a Service provider emphasizes two of its critical characteristics: a) The component is an independent executable entity that is defined by its interfaces and b) The services offered by a component are made available through an interface and all interactions are through that interface. The component interface is expressed in terms of parameterized operations and its internal state is never exposed. It is evident that the eventual view of Sommerville [63] with regard to a component is that components have two related set of interfaces which reflect the collection of services that the components provides and the collection of services that components requires to operate correctly.

4.2 Proposed definition of component

Over the last few decades, many definitions have been given about Components by practitioners and researchers. A few of those definitions are discussed in section 4.1. These definitions have evolved by
the observation of design principles that architects adopt and actions that they take when engineering Component based Systems. Some define Components as a uniquely identifiable modular part of a System that encapsulates its content. Some others define Components as a reusable Software element with a specification that can be combined with other Components to form an application. Some others define Component as a package that encapsulates a set of related functions or data. Some others define Component as a Software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. Some others define Component as a unit of composition with contractually specified interfaces and explicit context dependencies only. Some others define Component as an identifiable part of a larger program that provides a particular function or a group of related functions. It is evident that there is no common consensus about these definitions, but each of these definitions have been practically found to be appropriate for a specific situation. In this section, a definition of components is proposed.

![Component: Process Step](image)

**FIGURE 4.1: ILLUSTRATION OF A COMPONENT**

Generally, Components refers to the fundamental building blocks of systems and it is by putting together a logical static structure of components that the system architecture is realized. This thesis considers components as the general abstraction based on which the software system elements of a software architecture is defined. It is a notional environment that lies within the confines of the software system, and represents the most fundamental element of this system. In computing, information is the only resource that can be used to build software systems; this thesis considers components as containers of information and their operations. It is a symbolic name for (or reference to) information that a software
system manipulates. While a simple Component would encapsulate information, a set of access methods and transformative operations, a composite Component would encapsulate information, a set of access methods, transformative operations and other Components that are part of a composition. The represented information can change but the access methods and the operations remain the same over the time period of existence of the component. Components exhibit state and behaviour wherein state corresponds to the state of the information resources and behaviour corresponds to the operations that are permissible on these information resources.

Towards understanding this particular viewpoint, a simple example of a component designed to simulate a Process Step is illustrated in Figure 4.1. A process step is the most granular level of a process and is the building block of a process specification. It is a specification of a unit of work that is assigned to a role player. Without process steps, there can’t be a process because nothing is specified. Each process step is represented by a symbol that corresponds to the type of the process step. There are many types of symbols that signify the various task types that are expressed as the process step. Each step contains a specification for the information and the resources required, pre-conditions that must be satisfied before the role player can perform the step, and post requisites to validate whether the task identified by the step is completed successfully. Each process step is associated with a task and might be associated with additional information about the task or to indicate control flow within the step.

4.3 Review of Literature on Specifying Components

The objective of specifying components is to assist the understanding of the component’s essence and key properties pertaining to its structure, behaviour, Composition and operation and reason about it. Traditionally, a component model can provide a practical means of specifying components. Even though Component models are the basis of component frameworks, there is no universal agreement on what a component model should represent with regard to the component, what a component model is and what should be expressed by the component model. The general agreement is that component model is a schema that is adhered by a component so that a certain type of composition as defined by the designer of the component model can be supported. Another way of specifying components is by utilizing an Architecture description language which serves as a conceptual model for describing Architectures. It is evident that the distinction in the expression of components between Architecture description languages and Component models is very blurred. In this section, the different approaches to specify components is discussed.
According to Allen [44-45], in the WRIGHT Architecture description language, a component is defined by a component type description which has two parts an interface and a computation that the component performs. An interface is defined as a collection of ports with each port representing an interaction in which the component participates. A port specification describes two aspects of the components interface; the components intended behaviour and the expectations about the System within which the component interacts. The ports act as the partial specification of the components behaviour. The computation of the component is a full specification defined in terms of a series of operations/actions that the component is supposed to perform and it corresponds to carrying out the interactions described by the ports upon which the analysis of the components properties can be based. It is evident that in WRIGHT, the component specification provides an abstraction of the components actual behaviour comprising the ports, interfaces, signatures of the interfaces and the dynamics of the interaction.

According to Garlan et al [53], in the ACME Architecture description interchange language, a component represents computational elements and data stores of a System and is defined in terms of interfaces which are termed as ports. Components are used to specify elements of a variety of different computational models at varying levels of abstraction. Components expose their functionality through their ports. Each port identifies a point of interaction between the component and its environment. Usually, there are multiple ports corresponding to different interfaces of the component. A port corresponds to a set of operations/requests available/processed on a component. A component specification also includes properties which can be used to describe certain aspects of the computational behaviour or structure. It is evident that in ACME, Component specification is terms of ports, interfaces, operations and properties.

According to Feiler et al [60], in the architecture Analysis and Design Language, Components form the central modelling vocabulary. Components are specified by a unique identity (name) and are declared as a type and implementation within a particular Component category which defines the run time essence of the component. The component type declaration establishes a component’s externally visible characteristics and it consists of a defining clause and descriptive sub-clauses. It further defines features which are the interfaces of the component, flows which are the abstract channels of information transfer and properties which are the intrinsic characteristics of a component. The component implementation specifies an internal structure in terms of sub-components, interactions among the features of these sub-components, flows across a sequence of sub-components, modes that represent the operational states.
and the properties. It is evident that Fieler et al, support specification of components through a schema of specific set of software and hardware abstractions.

According to Lau et al [91-93], the interface of the component is the only point of access to the component. As a result, the specification of a component is therefore the specification of its interface which consists of precise definition of the components operations and context dependencies. Their Component specification is in terms of a context, restricting constraints, supporting interface and implementation as code. The context of a component is composed of a signature (containing sort symbols, function declarations and relation declarations) and a finite or recursive set of axioms. In an open context, some of the parameters in the signature need to be instantiated based on strictly defined constraints which are expressed as a closure. The interface of a component is defined in the context of the component in terms of operations (which are represented using logic programs) and context dependencies (which are the global parameters used in the signature of the component, the local parameters of the operations and the constraints in the context). The code of a component is in the form of binary and is instantiated before execution. It is evident that even though Lau et al try to put together a formal specification of a component, their specification is constrained by the presence of code/program (which is actually the realization of the component specification).

According to Sommerville [63], an essential part of the component specification is its interface specification which is usually defined as a set of abstract data types. Further, an interface specification comprises of the specification name followed by a set of generic parameters (which are abstract types of collections of other types); an introduction part that declares the sort (the type name) of the entity being specified; a description part where the operations are described formally using an unambiguous syntax and semantics for the type operations; the signature part defines the syntax of the interface comprising the names of the operations, the number and sorts of their parameters and the sort of operation results; and the axioms part that defines the semantics of the operations by defining a set of axioms that characterise the behaviour of the abstract data type. It is evident that Sommerville’s formal specification of a component comprises essentially of a signature part for defining the operations and its parameters and an axioms part where the relationships between the operations are defined.

According to Bruneton et al [84-85], in the FRACTAL Component model a component is described and understood as being composed of a membrane, which supports interfaces to its internal features, and a content, which consists of a finite set of other Components called sub-Components. The membrane has internal and external interfaces. External interfaces are accessible from outside the component, while
internal interfaces are only accessible for the component’s sub-Components. The membrane is composed of several controllers and provides an explicit connected representation of the component’s sub-Components. Each controller controls the behaviour of the components in terms of suspending, checkpointing and resuming activities of these sub-Components. Controllers also play the role of interceptors, a mechanism by which the external interface of a sub-Component becomes the external interface of the parent Component. The FRACTAL Component model does not enforce a pre-determined set of membranes and hence it is considered as an open Component model. However, the specification identifies categories of membranes and controllers that correspond to the nature of capabilities to be supported by the components. It is evident that while the FRACTAL Component is specified as comprising a set of interfaces; a set of controllers to manipulate these interfaces; and a collection of sub-Components that utilize these controllers and interfaces to provide the desired capabilities for the component.

According to Shaw et al [55], in their UNICON Architecture description language, Components are specified by a type, a specification and an implementation. The specification defines the units of association used in System Composition and the implementation can be primitive or composite. Essentially, each Component has an interface specification that defines its properties. These properties include the component’s type or sub-type, functionality, performance characteristics, and so on. The specific named entities visible in a component’s interface are its players. The interface includes the component type, assertions and constraints that apply on the entire Component, the signature, functionalities and interaction properties of its players. Further, the component type expresses the general class of functionality to be provided by the component. According to Shaw et al, Components in UNICON may be primitive or composite. While primitive Components are implemented as code in a conventional programming language, composite Components are defined as configurations of components and connectors. It is evident that Shaw et al view Components as predominantly computational entities that are specified using their interfaces.

According to Oquendo [86], in the π-Architecture description language, Components are specified in terms of external ports and internal behaviour. The architectural role of components is to specify the computational elements of a software system and the specific focus is on the computation to deliver System functionalities. Ports are described in terms of connections between a component and its environment. The architectural role of ports is to put together connections that provide an interface between the component and its environment. Connectors are special purpose Components that are specified in terms of external ports and internal behaviour. The architectural role of these special purpose
Components is to connect together Components. Behaviours uses these special purpose Components (at ports) to connect and transmit values between two Components. It is evident from this that, Oquendo views Components as providing the locus of computation and specified using ports and behaviour.

In CORBA [82], a component is a basic meta-type that is an extension and specialization of the object meta-type. Component types are specified in an Interface description language. A component declaration describes an interface for a component. This declaration specifies the name of the component, a list of interfaces that the component supports, single inheritance from other Component definitions and it may include in its body any attribute declarations together with declarations of facets and receptacles of the component, and the event sources and sinks that the component defines. It is evident that in CORBA, the definition of a component is predominantly an extension and specialization of an interface definition that encapsulates its internal representation and implementation.

4.4 Summary

This thesis considers software architecture as a static structural composition of components, wherein, components are analogous to “containers of information and their associated operations”. The nature of components is understood by how they work together in realizing the function. Each component affects behaviour of software as a whole and has variety of attributes carrying numerous complex processes and structures packaged in a discrete form. This thesis takes cognizance of the fact that components operate independently but are designed, constructed, and operated as an integral part of software and that different components coordinate and coexist together to manifest properties of software which correlate to its behaviour.