

Modeling Software Architecture with UML

Ramesh Ponnala¹, Gangadhar Adepu², Santhosh Kumar Nallavelly³

¹ Asst. Professor, Dept. of Computer Science & Engineering,
Sree Chaitanya Institute of Technological Sciences, Andhra Pradesh, India
ramesh.ponnala@gmail.com

² Asst. Professor, Dept. of Computer Science & Applications,
Sree Chaitanya Institute of Management & Computer Sciences, Andhra Pradesh, India
gangadhar24mca@gmail.com

³ Asst. Professor, Dept. of Computer Science & Applications,
Sree Chaitanya Institute of Management & Computer Sciences, Andhra Pradesh, India
santhosh1714@gmail.com

Abstract: *Software Architecture is being widely used today to describe a very high level design methodology of large & heterogeneous software systems. A good Architectural representation scheme holds the key to the effectiveness of a Software architecture description and usage. In this paper, we look at UML (unified modeling language) as a prospect for a generalized architecture description language. UML also "unifies" the design principles of each of the object oriented methodologies into a single, standard, language that can be easily applied across the board for all object-oriented systems and a scheme AND-OR DFD method is introduced and developed.*

Keywords: Software Architecture, Unified Modeling Language, Software Architectural modeling view

1. Introduction

An Architectural Style defines a family of systems in terms of a pattern of structural organization. An awareness of these Architectural styles can simplify the problem of defining system architectures. However, most large systems are heterogeneous and do not follow a single architectural style. Software Architecture determines how system components are identified and allocated, how the components interact to form a system, the amount and granularity of communication needed for interaction, and the interface protocols used for communication among stakeholders: Customers, managers, designers, programmers. Software Architecture consists of components, connectors, data, a configuration, and a set of architectural properties.

An important feature of architecture is the ability to facilitate development of large systems, with components and connectors of varying granularity, implemented by different developers, in different programming languages, and with varying operating system requirements. [1]

1.1 Component:

A component is an abstract unit of software that provides a transformation of data via its interface. Components can be computation units or data stores. According to [2], components are loci of computation and state.

1.2 Connector:

A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components. The connectors play a fundamental role in distinguishing one architectural style from another and have an important effect on the characteristics of a

particular style [3].

1.3 Datum:

A datum is an element of information that is transferred from a component, or received by a component, via a connector.

1.4 Configuration:

A configuration is the structure of architectural relationships among components, connectors, and data during some period of system run-time.

2. Literature Review

UML was created by Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997. OMG is continuously putting effort to make a truly industry standard. [4] UML stands for Unified Modeling Language.

- UML is different from the other common programming languages like C++, Java, and COBOL etc.
- UML is a language to specify, to visualize and to build and to document the artifact of the software systems, as well as to model business and other systems besides software systems. [4]

UML is a pictorial language used to make software blue prints. UML (Unified modeling language) is a clear and concise modeling language without being tied down to any technologies. It provides the ability to capture the characteristics of a system by using notations and is the language that can be used to model systems and make them readable. UML provides a wide array of simple, easy to understand notations for documenting systems based on the

Object-Oriented Design principles. These notations are called the nine diagrams of UML.

2.1 UML Diagrams

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). A diagram represents an elided view of the elements that make up a system. The UML includes nine such diagrams.

2.1.1 Use case Diagram:

This diagram is used to identify the primary elements and processes that from the system. The primary elements are termed as "actors" and the processes are called "use cases".

2.1.2 Class Diagram:

A class diagram shows a set of classes, interfaces, and collaborations and their relationships. This diagram is used to refine the use case diagram and define the detailed design of the system. The class diagram classifies the actors defined in the use case diagram into a set of interrelated classes. The relationship or association between the classes can be either an "is-a" or "has-a" relationship. Each class in the class diagram may be capable of providing certain functionalities. Class diagrams address the static design view of a system.

2.1.3 Object Diagram:

An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

The object diagram is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time.

2.1.4 State Diagram:

A state chart diagram shows a state machine, consisting of states, transitions, events, and activities. State chart diagrams address the dynamic view of a system. Objects in the system change states in response to events. In addition to this, a state diagram also captures the transition of the object's state from an initial state to a final state in response to events affecting the system.

2.1.5 Activity Diagram:

Activity diagram is used to capture the process flows in the system. Similar to a state diagram, an activity diagram also consists of activities, actions, transitions, initial and final states, and guard conditions. Activity diagrams address the dynamic view of a system. They are especially important in

modeling the function of a system and emphasize the flow of control among objects.

2.1.6 Sequence Diagram:

A sequence diagram represents the interaction between different objects in the system. This means that the exact sequence of the interactions between the objects is represented step by step. Different objects in the sequence diagram interact with each other by passing "messages". Sequence diagrams are called interaction diagrams in UML, which emphasizes the time-ordering of messages.

2.1.7 Collaboration Diagram:

A collaboration diagram groups together the interactions between different objects. This diagram helps to identify all the possible interactions that each object has with other objects. Collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

2.1.8 Component Diagram:

The component diagram represents the high-level parts that make up the system. This diagram depicts, at a high level, what components form part of the system and how they are interrelated. It also depicts the components called after the system has undergone the development or construction phase. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

2.1.9 Deployment Diagram:

The deployment diagram captures the configuration of the runtime elements of the application. This diagram is by far most useful when a system is built and ready to be deployed. A deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of the architecture.

2.2 Architectural Modeling Views

To describe Software Architecture, we use a model composed of multiple *views* or perspectives. In order to eventually address large and challenging architectures, the model we propose is made up of six main views:

- Logical view, which is the object model of the design (when an object-oriented design method is used)
- Process view, this view deals with concurrency and distribution, system integrity, and fault tolerance [5].
- Component view, which shows the grouped modules of a given system, modeled using the component diagram.
- Development view, which describes the static organization of the software in its development environment.
- Physical view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect.

- Execution view, which is the runtime view of the system. It involves the mappings of modules to runtime images, defining the communication among them, and assigning them to physical resources. Resource usage and performance are key concerns in the execution view.

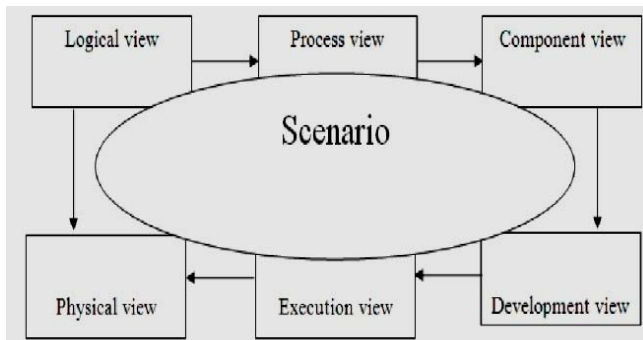


Figure 1: "6+1" View Modeling a System's Architecture

3. Experimental Results: C2 Generator

Let us consider a software system called C2 Generator. This software system would be written in an object oriented language like JAVA and it attempts to generate an architectural representation diagram based on the C2 Generator architecture [6].

It takes as input the components of the system to be modeled, the connectors and a list of who notifies whom. But it will suffice to say here that C2 Generator is an architecture description language (ADL) that is used to model user interface intensive software systems i.e., applications that have a graphical user interface (GUI) aspect.

This architectural style consists of components and connectors. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector. The bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a single connector.

In C2-style architecture, connectors transmit messages between components, while components maintain state; perform operations, and exchange messages with other components via two interfaces which are called top and bottom.

Each interface consists of a set of messages that may be sent and a set of messages that may be received. Inter-component messages are either requests for a component to perform an operation, or notifications that a given component has performed an operation or changed state.

In the C2 style, components cannot interact directly but can do so using the connectors. Each component interface can be attached to at most one connector. A connector, however, can be attached to any number of other components and connectors. Request messages can only be sent "upward"

through the architecture, and notification messages can only be sent "downward."The C2 style has another requirement that the components communicate with each other only through message passing and never through shared memory. Also, C2 requires that notifications sent from a component correspond to the operations of its internal object, rather than the needs of any components that receive those notifications.

This constraint on notifications helps to ensure substrate independence, which is the ability to reuse a C2 component in architectures with differing substrate components (e.g., different window systems). The C2 style explicitly does not make any assumptions about the language(s) in which the components or connectors are implemented, whether or not components execute in their own threads of control, the deployment of components to hosts, or the communication protocol(s) used by connectors.

There are four primary components in this software. The CreateConnection component has five subcomponents, which are the various steps taken to create a connection. First, the component to be connected to first created component is identified from the connection list. Then new ports are created and attached to both these components. We assume here for simplicity that both components can have unlimited number of ports and so unlimited number of connections. Then the connector is created and the two ports are connected. It is obvious that the steps in creating a new connection start with reading a component name from the connection list till the connector is attached to the two newly formed ports. This whole process has to be repeated till there are no more entries in the connection list. This iterative property of the system cannot be known from the decomposition model, though it must occur if the system executes correctly. Second, there might be repeated entries in the connection list.

Table 1: Process Decomposition of C2 Generator

Module Name	Submodule(s)
1) ReadInput	
2) CreateComponent	
3) ReadConnectionList	
4) CreateConnection	CreateComponentToBeConnected
	CreatePorts
	ConnectPortsToBothComponents
	CreateConnector
	ConnectBothPortsWithConnector

There is no restriction to the number of connections one component can have with other components. For an entry that refers to a component which has already been created, one doesn't need to create it again, but just identify that component and create a new port. Hence, once an entry has been read from the Connection List, one of two things happen depending on the read value. Either the component doesn't exist and needs to be created, or it exists and needs to be identified. Again, there is no way of knowing this from the decomposition model. Let us now consider how the AND-OR DFD tackles these issues.

3.1 AND-OR DFD Representation of C2 Generator

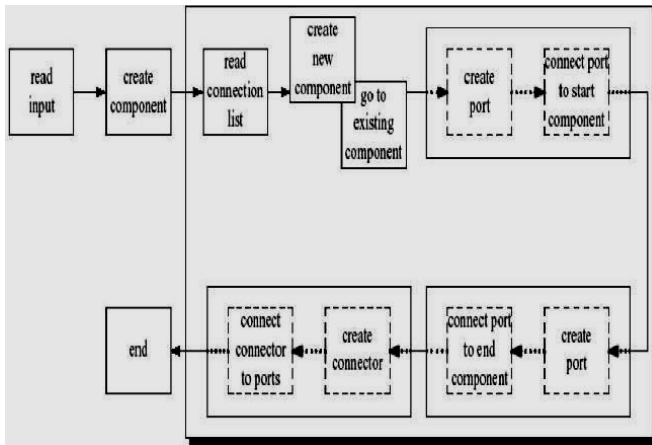


Figure 2: AND-OR DFD representation of C2 Generator

From Figure 2, we see that the data flow is represented by this modified DFD, but with two significant differences. Firstly, there is an OR-group of two components that illustrate that once an entry has been read from the connection list, either a new component is created, or control moves to an existing component, depending on the value read from the connection list. Second, the iterative portion of the system has been illustrated by a shaded box. So we now can tell that the steps starting from the reading of the connection list to the connection of the ports by a connector are iterative and are executed for each entry in the connection list

3.2 Architecting UML of C2 Generator

In order to see how UML can construct the Software Architecture of a system, let us go back to the example of the C2 Generator. Table 2 shows the logical decomposition of the system. The use of layering in modeling C2 style architecture for GUI intensive software systems [6] and the use of layering in representation of module view of an architecture using UML also indicate the vast potential for the layering style.

The logical (conceptual) decomposition highlights the main components of the system and their subcomponents if any.

Table 2: Logical Decomposition of C2 Generator

Module Name	Sub Module(s)
1) C2 Generator	CreateComponent CerateConnector UpdateComponentList UpdateConnectionList
2) Component	CheckForFreePort CreatePort
3) Port	CreateConnector
4) Connection	UpdateConnectingComponent UpdateConnectedComponent

We see that the C2Generator component has the task of creating the component(s) and the connector, and updating the component and connection lists. The component module checks for free ports on the component(s) and if there are free ports, then it creates the physical port. The Port component creates the connector in turn, and the connector component joins the two components (called the connecting component and the connected component here) and updates the two components for the connection created.

3.3 Conceptual View of C2 Generator

From table2, we came to know about Logical decomposition of C2 Generator. Let us now try to construct the logical architectural view for C2 Generator. Figure 2 shows the conceptual architectural view of the C2 Generator using UML constructs.

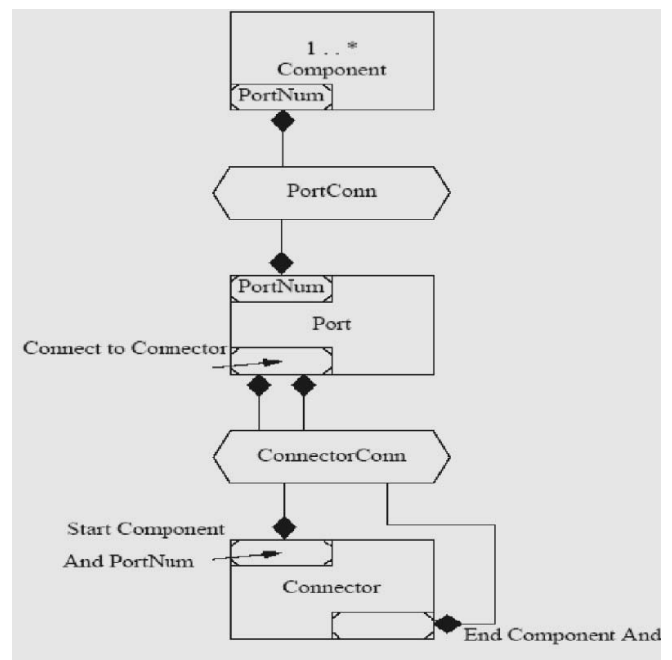


Figure 3: Conceptual view of C2 Generator

Figure 3 show the conceptual architectural view of the C2 Generator using UML constructs [7]. The problem with this

representation lies in the relationship of the port and the Connector Conn. i.e. a connector can be broken off from one component and joined to another component. So a composition doesn't hold good here. Even an aggregation doesn't hold good because when the connector is isolated from the ports of both the connecting components, it ceases to exist independently. So here is a situation where there is a composition relationship that involves two components and a connector.

3.4 Execution View of C2 Generator

The execution view or process view of C2 Generator will be modeled from the process decomposition diagram we saw earlier. Figure 4 shows one sequence diagram representing the execution configuration of the C2 Generator.

The C2Generator first creates the connecting component by calling the Create Component () procedure and interacting with the component module. The component module in turn then creates a port and connects the newly created component to it by calling the Create And Connect To Port () procedure and communicating with the Port module. The Port module now creates the connector and attaches the port to this connector by calling two functions and talking to the Connector module. Once this is done and the control is back to the C2 Generator component, it now reads the connection list and checks if the component to be connected exists or not. If it exists, control moves to this existing component and that component is connected via a new port to the already created connector. If the component doesn't exist then it is created before being connected to the connector.

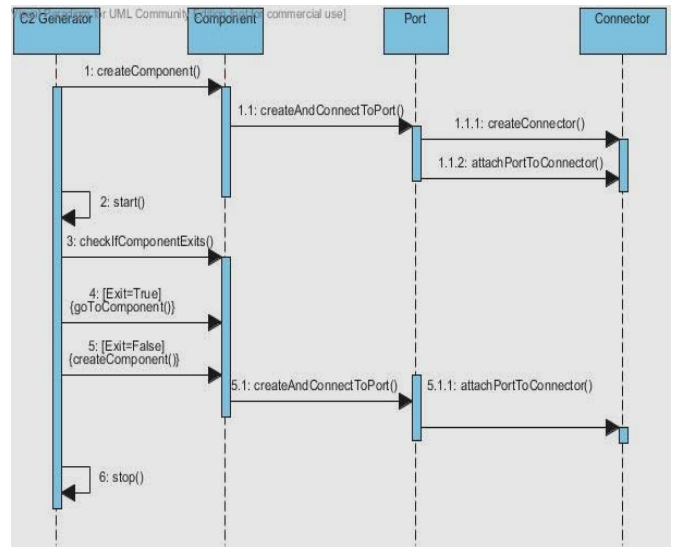


Figure 4: Sequence Diagram for C2 Generator

This implementation is efficient because the control flow doesn't move back and forth. Both the components are ready before the ports are created and both the ports are ready before the connector is created and the connection made. So we see that UML is rather useful for representing different views of the software architecture of a system [7], [8]. It does reasonably well and represents all the facets of that view clearly. Moreover, UML is good for all the views, and not just the process view which can be adequately represented by the AND-OR DFD. Moreover, we can extend UML by constraints, tagged values, stereotypes and profiles [9].

4. Summary

Table3: Summary of C2 Generator

View	Components	Connectors	Containers	Stakeholders	Concerns	Tool Support
Logical	Class	association, inheritance, containment	Class category	End-user	Functionality	Rose
Process	Task	Rendezvous, Message, Broadcast, RPC, etc.	Process	System designer, integrator	Performance, availability, S/W fault tolerance, Integrity	UNAS/SALE DADS
Component	Module	Interaction	Component	Developer	Interoperability	Rose
Development	Subsystem	compilation dependency, "with" clause, "include"	Subsystem (library)	Developer, manager	Organization, reuse, portability, line of-product	Apex, SoDA
Physical	Node	Communication medium, LAN, WAN, Bus, etc.	Physical subsystem	System designer	Scalability, performance, availability	UNAS, Openview DADS
Execution	Mappings of node		Run time view	End-user, Developer	Resource usage and performance	Rose
Scenario	Step, Scripts		Web	End-user, developer	Understandability	Rose

References

- [1]. Medvidovic, N., Taylor, R.: "A framework for classifying and comparing architecture description languages," Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, Zurich, Switzerland, Pages: 60 - 76, 1997.
- [2]. Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G.: "Abstractions for Software Architecture and Tools to Support Them," IEEE Transactions on Software Engineering, 21(4):314-335, April 1995.
- [3]. Perry, D., Wolf, A.: "Foundations for the study of software architecture," ACM SIGSOFT Software Engineering Notes, Volume 17, Issue 4 (October 1992), Pages: 40 - 52, 1992.
- [4]. "OMG Unified Modeling Language Specification," Version 1.5, March 2003. Clements, C., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.
- [5]. "Documenting Software Architectures: Views and Beyond." Addison- Wesley, 2003, ISBN 0-201-70372-6.
- [6]. Medvidovic, N., Rosenblum, D.: "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures." In Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), pages 161-182, San Antonio, TX, February 22-24, 1999.
- [7]. Hofmeister, C., Nord, R., Soni, D.: "Applied Software Architecture," Addison- Wesley, 2000, ISBN 0-201-32571-3.
- [8]. Hofmeister, C., Nord, R., Soni, D.: "Describing software architecture with UML," Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), Pages: 145 - 160, 1999.
- [9]. Medvidovic, N., Rosenblum, D., Robbins, J., Redmiles, D.: "Modeling Software Architectures in the Unified Modeling Language," ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11, Issue 1 (January 2002), Pages: 2 - 57, 2002.

Author Profile

Ramesh Ponnala received the MCA and M.Tech degrees in Computer Science and Engineering from Jawaharlal Nehru Technological University. He is Sun Certified Java Professional and AP-SET Holder. From 2009 onwards he is staying in Sree Chaitanya Institute of Technological Sciences as an Assistant Professor.

Gangadhar Adepu received the MCA and pursuing M.Tech degrees in Computer Science and Engineering from Jawaharlal Nehru Technological University in 2006 and 2012, respectively. During 2006-2010, he stayed in Sree Chaitanya Institute of Management & Computer Sciences as an Assistant Professor.

Santhosh Kumar Nallavelly received the MCA and pursuing M.Tech degrees in Computer Science and Engineering from Jawaharlal Nehru Technological University in 2006 and 2012, respectively. During 2006-2010, he stayed in Sree Chaitanya Institute of Management & Computer Sciences as an Assistant Professor.