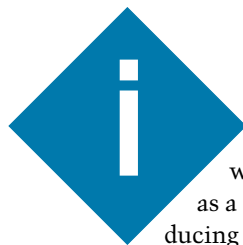


## FEATURE ARTICLE

Venu Kosuri

# UML in a Product's Life Cycle

Intended to be an introduction to UML, this article focuses on illustrating how to use its concepts in the development life cycle of a product. UML is a state-of-the-art modeling methodology useful for real-time systems, so if you're still a beginner, Venu will guide you through to the end.



I wanted to write this article as a means of introducing UML, which is a state-of-the-art, object-oriented modeling methodology, to beginners. UML combines the current good practices of dozens of practitioners into a cohesive approach useful for standard desktop, client-server, and real-time systems.

My aim is to briefly explain UML concepts and to illustrate how these concepts can be used in the development life cycle of a product. I won't go into much detail about notational conventions of various UML diagrams, as these conventions are described in brief wherever necessary. Mainly, I wish to impart the sense of using UML in product development.

### BASIC VIEWS

UML views a system in five different views—use case, structural, behavioral, implementation, and environmental (see Figure 1)

Also, UML provides many dia-

grams to view a system. All the diagrams can be classified according to the above views (see Table 1).

### THE WHOLE PICTURE

Use case diagrams are often used for gathering requirements of the product. After this aspect of the system is defined, evolution begins. Of all the requirements, some are implemented in hardware and some are implemented in software. This is basically decomposing the system into hardware and software partitions. This is an entry point for the development of software as well as hardware. After the software system is defined, then decomposition of the software system can take place, which is nothing but domain classification.

It is important to understand the dynamic behavior of the system after structuring the system. In order to capture this, sequence diagrams are used to anticipate all the scenarios that the system is expected to go through. For complex systems, the state approach is also used to get the clear distinctions of the system's states and related scenarios. Understanding thoroughly the dynamic behavior of all domains is an entry point to define the requirements of each domain and the general mechanism of communication

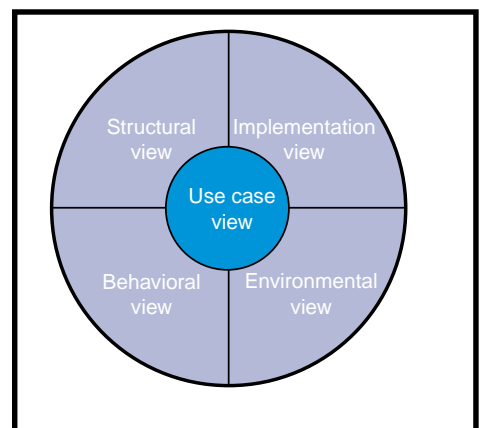


Figure 1—These are modeling views of a system (UML perspective).

View	Used for	Diagram(s)
Use case	Functional requirements	Use case
Environmental	Physical constraints	Deployment
Behavioral	Design refinement	Sequence, collaboration, activity
Structural	Design refinement	Domain, class, package
Implementation	Low-level design	Component, state

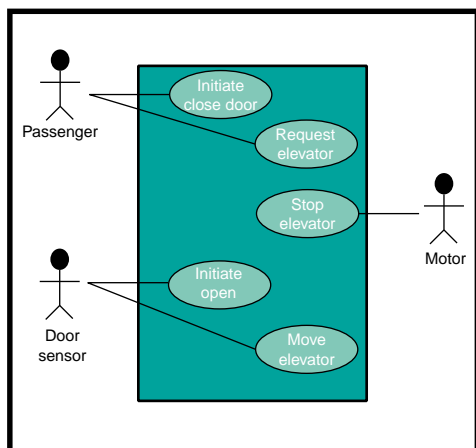
**Table 1**—Different views of a system are designed to aid in various stages of the development work, and each view of a system is best represented by certain diagrams.

among the domains.

Domain's requirement is to realize the domain. So, each domain is broken into a set of components, or modules. Primarily, all components act together to perform a desired requirement of the domain. Component diagrams capture this information. For real-time embedded systems, the port-based approach is most used. Each port defines a specific protocol that is responsible for communication among specified components. This component division is a starting point to understanding the component.

If the component is being developed with OO techniques, class diagrams, sequence diagrams, state diagrams, and collaboration diagrams will capture the essence of the component for implementation. Of course, whether or not all of these diagrams will be used is based on the complexity of the component. But often, just a class diagram is sufficient.

Suppose the component does not have any threads of its own (i.e., passive component). Package diagrams work best to decompose the component into subcomponents, which are easily implemented. Here, subcompo-



**Figure 2**—This is a sample of use cases for an elevator system.

nents are small entities that implement a specific functionality of the component.

Deployment diagrams and activity diagrams are rarely used. Activity diagrams are similar to flowcharts except for the fact that parallelism can be shown in activity diagram. Generally, if a method of class or a function is big, it's better to use activity diagrams. However, most developers use flowcharts. If the system is physically big, deployment diagrams are used to show physical layout.

Now, let us apply these diagrams and their relevance in a product or system evolution.

## USE CASE VIEW

In any product development, the first step is marketing. Marketing research results in commercial requirement specification (CRS) for a product. The next step is to gather functional requirement specifications (FRS), often called system requirements, of the product from CRS.

Use case view helps with gathering FRS. In this view, a system is modeled as a black box. Any real-time embedded system responds to a large number of external events that cause a system to do some desired activity. Each desired activity is nothing but a functional requirement. So, system requirements can be better understood by understanding the actors that generate external events and the work that must be carried out in response. These event-response pairs are called "use cases."

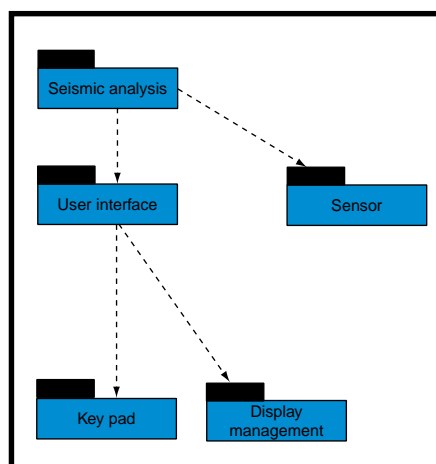
The two steps involved in use case analysis are to identify the actors and the use cases. An actor is anything that interacts with the system and, thus requires something from it. An actor can be like a human, machine, independent sys-

tem, sensor, or so forth. For example, for an ATM, the insertion of a debit card is an action initiated by a human actor.

A use case describes anything the actors want the system to do. In an elevator control system, a person requests a lift to a particular floor, the motor control box requires the elevator to open the door when the elevator comes to a halt, and the door sensor detects an obstruction and prevents the door from closing.

## USE CASE DIAGRAM

UML provides a representation to capture all actors and use cases for a system. This representation is simply a



**Figure 3**—Take a look at the primary domains of a seismic instrument.

use case diagram. Figure 2 shows a system as a rectangle, use cases as ovals inside the rectangle (named according to the task to be carried out), and actors as stick figures (named by the role that the actor plays).

## STRUCTURAL VIEW

Use cases provide an external view of the system and its requirements. The next step is how to implement these requirements.

First, let's break the complexity of the system into manageable pieces called domains. A domain is a set of closely related classes or sub-domains that work together to provide services to other domains (e.g., a server domain provides services to another domain, the client domain). The services of one domain can be accessed through

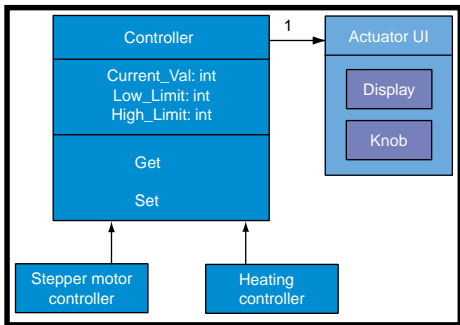


Figure 4—Here's a class diagram in an industrial scenario.

bridges. Thus, a bridge is communication coupling among domains. After the concept of domains and bridges is clear, the next step is to break the system into domains (i.e., domain modeling).

## DOMAIN MODELING

There are four steps in domain modeling—identify the application domain, identify key system functions, assign key system functions to domains, and write domain descriptions.

The application domain provides services specific to this particular system. Almost all systems have a single application domain from the end user's point of view. For example, the application domain of a microwave oven could be called microwave cooking.

Key system functions are high-level functions that the system must perform to satisfy the end users' requirements.

Group-related key system functions are assigned together into domains and a meaningful domain name is selected. Key system functions specific to this particular application are allocated to the application domain.

A description of the domain should be written that communicates its capabilities as well as the services it requires. Well-written descriptions improve model comprehension and reduce confusion. For example, the display management domain of an ATM could be set up in three steps (capabilities should be displayed in a text message via the display hardware, and there are no required domains that

depend on other domains).

First, build a domain chart diagram to show the domains identified in previous steps and their dependencies. UML provides package diagrams to represent a domain chart. In package diagrams, domains are represented as tabbed rectangles and bridges by dependencies as dotted arrows. Place the application domain at the top of the domain chart. A typical domain chart for a seismic instrument looks like Figure 3.

Next, classify domains as analyzed or realized. Some domains may exist from other systems or products. In that case, reuse of the domain is sufficient. Some domains may be new ones and some domains may not be developed with OO analysis.

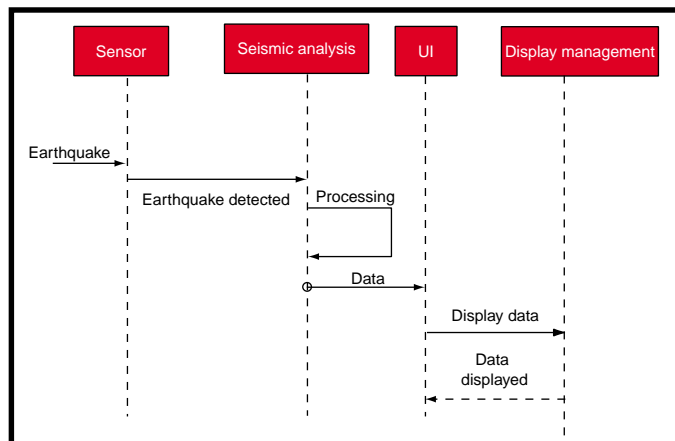


Figure 5—Here we have auto detection of an earthquake scenario by a seismic instrument.

And finally, conduct a review of the domain chart with application experts and team members. Reviewers should check if all key system functions are assigned to a domain, if all names and descriptions are clearly written, and if there are any missing bridges.

## CLASS DIAGRAM

Class diagram is the core of OO modeling. It represents the core classes of a domain or a component (which I'll define later in the article) showing the classes, attributes, and relationships owned by the domain or the component that is being

modeled. Basically, class diagrams are shown in three perspectives—conceptual, specification, and implementation.

In the conceptual view, the class diagram represents the concepts in the domain under study. These concepts will naturally relate to the classes that implement them, but often there is no direct mapping. Indeed, a conceptual model should be shown with little or no regard for software that might implement it, so it can be considered language independent. This is useful when analyzing the domain.

In the specification view, the class diagram is drawn to represent the interface of software, not implementation. Thus, you are looking at types rather than classes. OO modeling puts

a great emphasis on the difference between interface and implementation, but this is often overlooked because the notion of class in an OO language combines both interface and implementation. Actually, a type represents an interface that may have many different implementations, because of the implementation environment, performance characteristics, or vendor. This is useful when analyzing a component (i.e., in the component specification stage).

In the implementation view, implementation aspects are shown. This is probably the most often used perspective, but in many ways, the specification perspective is often a better one

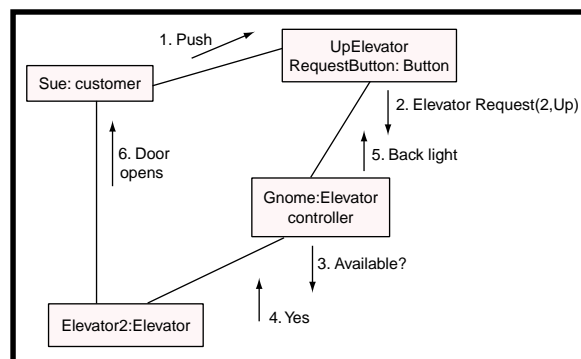


Figure 6—This is a collaboration diagram for a user's request for lift in an elevator system.

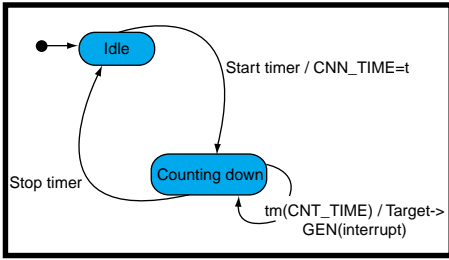


Figure 7—Here is a simple counter's state diagram.

to take. This is useful when implementing a component (i.e., in the design stage).

Understanding the perspective is crucial to both drawing and reading class diagrams. But, most designers don't take care to get their perspective sorted out when they're drawing.

In the class diagram (see Figure 4), classes are represented as simple boxes (such as Stepper Motor Controller class), compound boxes (such as Controller class), or boxes containing other boxes (such as Actuator UI class). The first two are semantically identical.

Simple boxes are classes where the internals of the class aren't displayed. The compartmentalized box lists some or the entire attributes (middle compartment) or operations (lower compartment). The third box type is a special kind of class called composite. It is different from other classes in that it oversees a group of objects that together act as a cohesive whole. The composite class is responsible for the creation and destruction of its component objects.

All of the boxes discussed thus far have been normal classes. Running programs will instantiate these to make one or more object instances of each. The relations among these classes are shown in Figure 4 (with arrows). There is a specific meaning for each arrow.

## BEHAVIORAL VIEW

So far, the system has been described structurally how it implements the desired requirements. Now, the most important concept is modeling its run behavior with each use case, the domains that are participating, and interaction of the classes within each domain.

For each use case, some domains

involve certain specific paths of interaction within a specific time, fulfilling the use case requirement. This specific path of interaction is termed a scenario. Each use case may involve more than one scenario. In modeling, the behavior of the system involves identifying all of these scenarios. To represent these interactions, UML provides two sets of diagrams, sequence diagram and collaboration diagram.

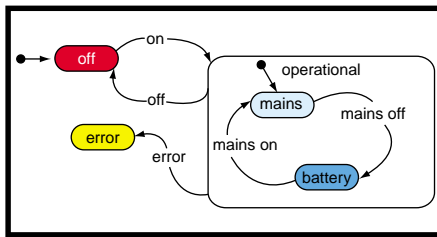


Figure 8—Here's a state chart for a computer engine that runs either by battery or mains.

## SEQUENCE DIAGRAM

The sequence diagram illustrates all the scenarios as per time scale. Scenarios at system level involve interactions among the domains. Similarly, scenarios at component level involve classes and the instances of the classes of the component. The sequence diagram of a seismic instrument for a particular use case is illustrated in Figure 5.

The use case is "auto measuring of earthquake intensity." The scenario is shown at system level. Here, every rectangle shows a domain or an object of a domain. The vertical line indicates the lifeline, which represents the object's life during the interaction. A lifeline for a domain doesn't make sense, but for an object, it is ex-

tremely relevant. The half-filled arrow represents an asynchronous message, a synchronous message is indicated by a filled arrow, and the response to a synchronous message is represented by a dotted arrow.

## COLLABORATION DIAGRAM

Generally for domains, a collaboration diagram doesn't make sense, because the collaboration diagram indicates the static relationship among all objects interacting for a particular scenario. Naturally this diagram is relevant for objects. Figure 6 shows relationships among objects in an elevator controller system. FSM

Another mechanism to catch dynamic behavior is the state machine. A finite state machine (FSM) is a mathematical model of a system that attempts to reduce the model complexity by making simplifying assumptions. It assumes the system being modeled can assume only a finite number of conditions (called states). The system behavior within a given state is always the same, and the system resides in states for significant periods of time.

The system may change these conditions only in a finite number of well-defined ways, called transitions, which are the responses of the system to

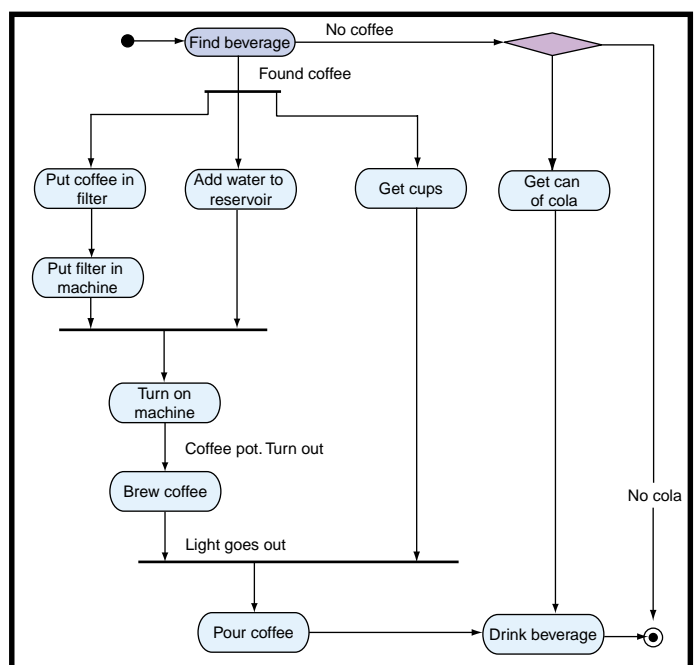
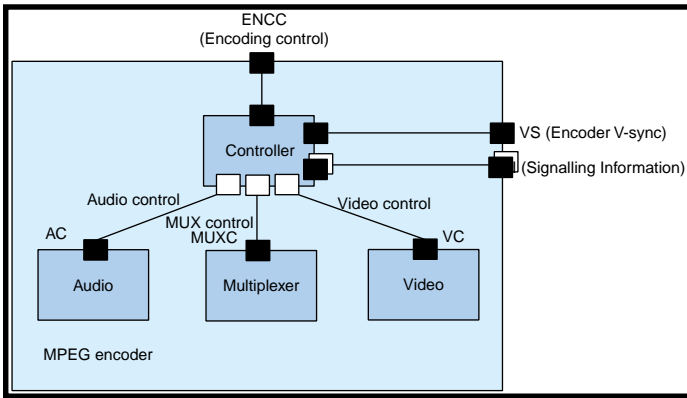


Figure 9—An activity diagram for getting a beverage is shown here.



**Figure 10**—This is a basic MPEG encoder. A rectangular box indicates a component. The connecting lines with open (not darkened) or closed (darkened) rectangles show protocols.

ber of states in each to model all conditions. This requires  $O(xn)$  states to model  $N$  state machines with an average of  $X$  states in each. Logically, it should be possible to model such a concurrent system in  $O(xn)$  states, a much simpler proposition.

## STATE CHARTS

State charts are an attempt to overcome the limitations of traditional FSMs while retaining their good features. State charts include both the notions of nested, hierarchical states, and concurrency while extending the notion of actions (see Figure 8). There are many forms of state charts available, each with slightly different semantics. The most popular one is a Harel state chart.

In a nutshell, state charts provide a powerful mechanism to model the behavior of a system. It can be used to model the dynamic behavior of a system, a domain, or an object of a domain.

## ACTIVITY DIAGRAM

Let's define an activity first. From a conceptual point of view, an activity is some task that needs to be done, whether by a human or by a computer. From a specification perspective or an implementation perspective, an activity is a method for a class. So, activity diagram represents a series of activities that need to be completed to do a particular activity (see Figure 9). It shows behavior with control structure. It also shows many objects after many uses, many objects in single-use case, and implementation of method. It can show the threads also, because parallelism can be shown graphically.

## IMPLEMENTATION VIEW

After domain classification, it is necessary to break the domain into components so that they can be implemented. Here, each component groups a set of activities to fulfill certain requirements of the domain.

For example, driver components provide the interface for the underlying hardware. Each component depends on some other components of the same domain or external domain. External domain communication is

events and take no time.

More precisely, a state is a distinguishable ontological condition that persists for a significant period of time. Transitions are responses to events that move the system from state to state. State diagrams describe the behavior of a system on the whole. There are many forms of state diagrams, each with different semantics. The state machine of a simple one-shot timer is shown in Figure 7. Such a timer is generally in two states, idle and counting down.

There are two kinds of approaches for representing a state diagram. A Mealy FSM associates actions with the transitions between states. A Moore FSM associates actions with the states themselves rather than the transitions.

## SCALABILITY LIMITATIONS

It is difficult to represent complex systems with FSM models. The methods work well for simple, state-driven systems, but don't work well for large systems. The scalability of FSM stems from two fundamental problems—the flatness of the state model and its lack of support for concurrency.

Flat state machines do not provide the means to construct layers of abstraction. All states are equally visible and are considered to be at the same level of abstraction. Consider a simple model of a car, which can be stopped, moving forward, or backwards. In a lower abstraction level, the pistons of the engine compress the gas in the firing chamber, expanding the gas in the firing chamber, filling the chamber with the gas/air mixture, or emptying it of exploded gas mixture.

If you do not arrange the states in a hierarchical fashion, then you must

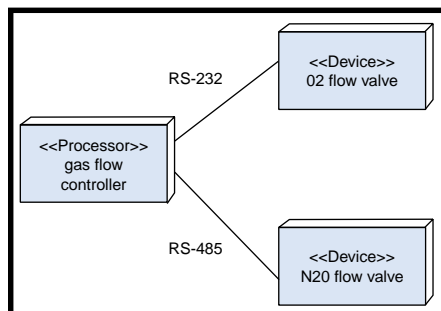
consider emptying the gas chamber at the same level of detail as moving forward, which it clearly is not. The state of moving forward in principle contains all the states of the pistons.

## LACK OF SUPPORT

Another serious problem with the traditional state machine is its lack of support for concurrency. This leads to a combinatorial explosion in the number of states to model. Consider a simple system that can be thought of as in one of four states—off, start-up, operational, or error. Also, it can run from either battery or main.

The fact that the system is in operational state is totally independent of whether or not it is running from battery or main. However, because traditional FSMs have no notion of independence, you must combine the independent states together. This yields a state like operational-battery and operational-main.

If you could model the FSM as two independent parts, the diagram would be simplified. This is called the combinatorial state explosion because the modeling of multiple concurrent FSMs requires the multiplication of the num-



**Figure 11**—A gas flow controller controlling oxygen and nitrogen dioxide in a chemical plant is illustrated here.

already captured in domain communications (i.e., bridge). All internal as well as external communications of a component can be specified in terms of protocols. Each protocol specifies a specific communication. Usually a component diagram captures this information (see Figure 10).

## ENVIRONMENTAL VIEW

In this view, the system is understood physically. The UML deployment diagram provides this view (see Figure 11). It shows the physical relationships among software and hardware components in the delivered system. Every component is shown as a node, and each node is represented as a cube. Connections among nodes represent communication paths over which the system components interact.

After having some idea about UML and its diagrams, you begin to realize the significance of each diagram in the life cycle of a product. 📄

*Venu Kosuri specializes in real-time embedded systems for consumer and defense electronics with Philips Software Centre, Bangalore. He holds a bachelors degree in technology for electronics and communication engineering from Regional Engineering College, Warangal, Andhra Pradesh, India. He may be reached at [venu.kosuri@philips.com](mailto:venu.kosuri@philips.com).*

*Special thanks to Rahul Khosla for reviewing this article.*

## RESOURCES

M. Flower, *UML Distilled: A Brief Guide to the Standard Object*

*Modeling Language*, Addison Wesley

Longman, Reading, MA, 1999.

B.P. Douglass, "UML Statecharts," *Embedded Systems Programming*, January, 1999.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, [subscribe@circuitcellar.com](mailto:subscribe@circuitcellar.com) or [www.circuitcellar.com/subscribe.htm](http://www.circuitcellar.com/subscribe.htm).