Test and Measurements System Modeling: Addressing the Root of the Problem

Filipe Altoe, PMP

Principal at TSXperts (www.tsxperts.com)

Introduction

The Universal Markup Language, UML, is a general purpose modeling language. It was introduced in 1997 by the Object Management Group, the OMG, with the original intention of providing information technology professionals with a stable and common language that could be used to communicate software designs amongst multiple software developers.

UML is a set of different types of diagrams, each of which can be used to represent a level of abstraction, or level of details, of a system. One can use a specific diagram type to model the up highest level of a system, and drill down into the details of each of the highest level components by utilizing other types of UML diagrams that are better suited to represent different levels of details. It offers a standard way for people to visualize the system blueprints.

UML version 2.2 has fourteen different types of diagrams that are available for use, broken down into two high level types: structure diagrams and behavior diagrams.

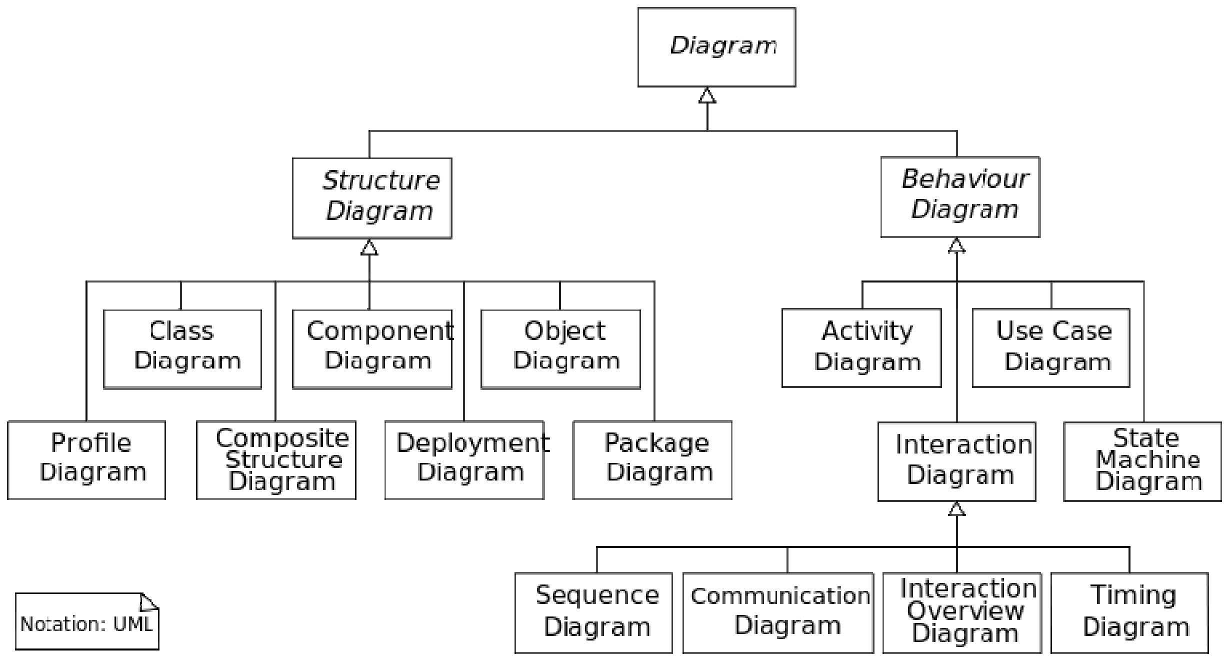

Figure 1 – UML 2.2 Fourteen Diagrams

Behavior diagrams emphasize what must happen in the system being modeled, or, the behavior of the system, or, why not say, the functionality (requirements) of the system. One can intuit that these types of diagrams would be good candidates to visually represent “What” the system must do. One may remember that the project “Whats” are related to the project objectives and project requirements.

Structure diagrams emphasize the things that must be present in the system being modeled; or, “How” the system will implement “What” the system needs to do. One can also intuit that these types of diagrams would be great candidates to model the system the system design, the “Hows” of the system being built.

In its core, one of the strongest points of UML is that it provides a visual representation of a set of functions in a non-technical manner. As we will see below, the diagrams present a visual language that is very intuitive for one to follow, at the same time that provides standardization, one of the features of any language.

Another clear advantage of UML is that a system can be fully modeled either by taking advantage of all fourteen different UML diagrams, or, it can also be fully represented through the utilization of just a subset of those diagrams. Simpler systems, with less interconnectivity between departments and with less overall stakeholders will yield a simpler overall UML representation than a very large and complex system. We will skip the detailed explanation of each of the fourteen diagrams and will focus on the ones that are directly used as a basis for the TMPM framework. Also, we will introduce each diagram at opportune times in the book, as they are needed as the foundation for the topic under discussion.

Behavior Diagrams and the Project Objectives and Requirements

This article will only cover the most used diagrams for the purpose of test and measurements systems modeling. We shall start by detailing two behavior diagrams; use case diagram and activity diagram. As it was mentioned previously, the behavior diagrams are the ones that are most useful in the modeling of project objectives and requirements.

Use case diagrams describe the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases. The actors represent elements that will interface/utilize the system, either human beings or other systems that are external to the system being modeled. The figure below shows a use case example. This use case example models the functionality of a portable audio player. One can notice that this system will have three main high level functions; operate audio player, maintain play list and maintain audio player, all of them to be performed by the Listener actor.

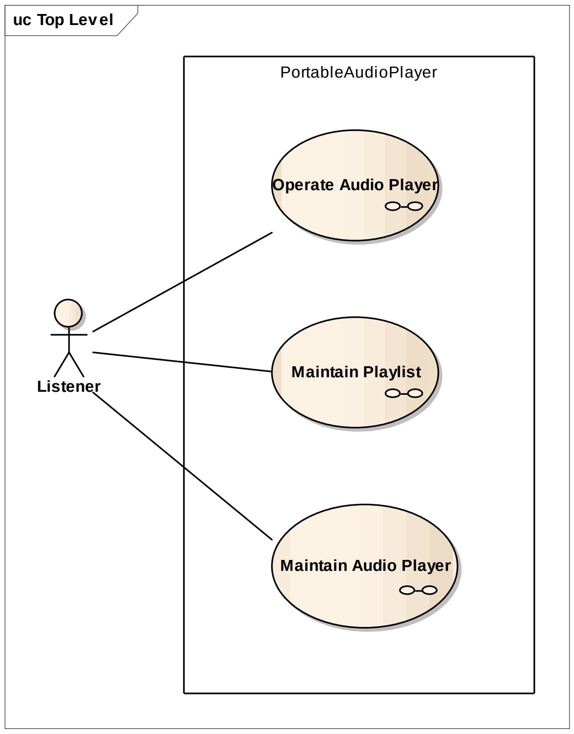

Figure 2 – Example of Use Case Diagram

The next natural activity to model this system would be to drill down and define the use cases that are involved in each one of those three high level functions. Figure below shows the use case diagram for the Operate Audio Player use case.

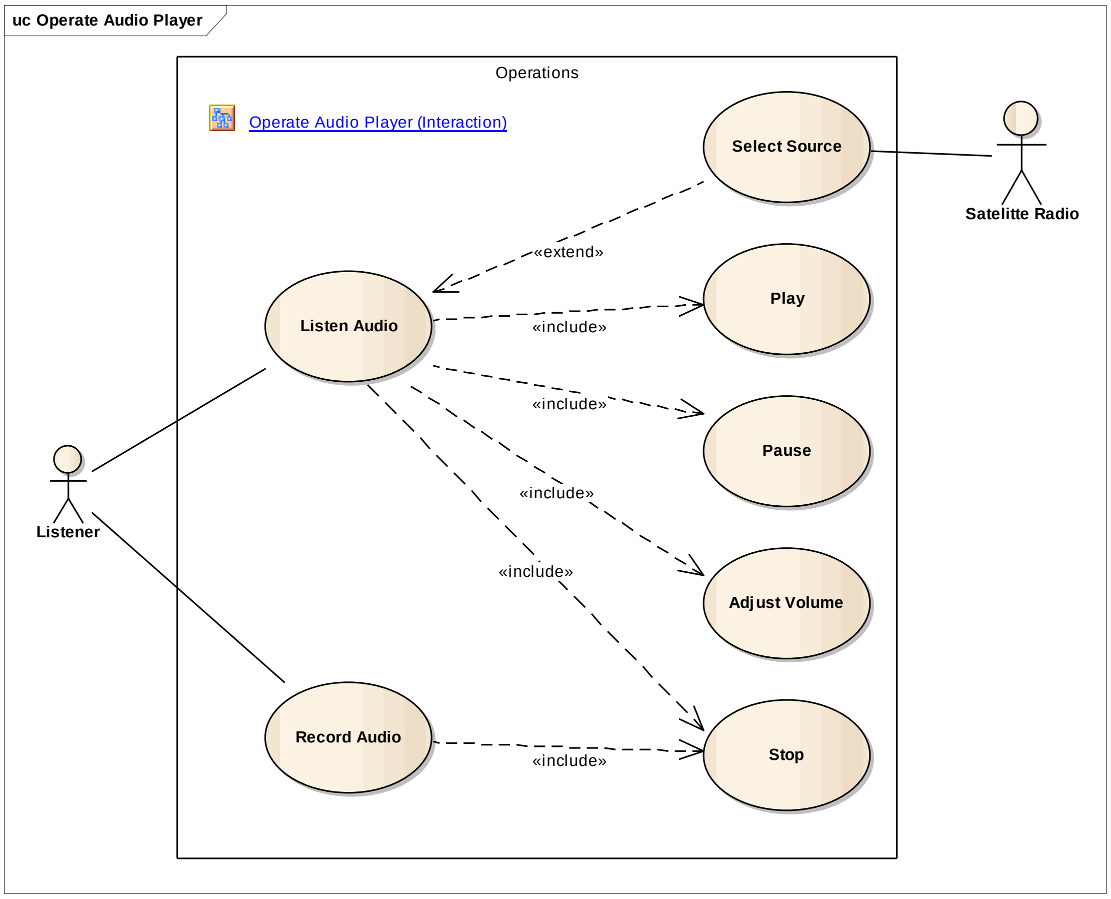

Figure 3 – Operate Audio Player Use Case

If the modeler would like to divide higher level use cases into a few more detailed ones, she should use one of the two available use case relations; include and extend. Include is used to separate logical sequence of actions from the higher level use case. The extend relation models optional actions from the higher level use case.

The way to interpret what is being modeled above is that there are two high level functions on the operate audio player use case; listen audio and record audio. When the Listener actor executes the record audio function, the record audio includes the action of being stopped; which will need to happen at some point. When the Listener actor selects listen audio, it will be played, paused, the volume be adjusted and stopped. Therefore, all of these other use cases are “included” in the two high level use cases.

Another way to look at the include relationship is that the lower level use cases can potentially be “included” by multiple higher level use cases as part of its functionality. For example, on the diagram above, the stop use case is used by both listen audio and record audio use cases.

The last use case to be explained is Select Source. Notice that when the Listener is executing Listen Audio, it has the option to select the audio source from MP3 list or satellite radio. However, the user doesn’t necessarily need to do that, as it will have a default selection. Therefore, this use case has an “extend” relationship with the Listen Audio use case. Also, notice how the Satellite Radio actor has been represented, since it is an external element to the system being modeled, and it may be accessed by the Select Source use case. The MP3 list, the other source the player can select hasn’t been represented as an actor as it is maintained within the scope of the audio player.

The use case diagram is, if not the first, one of the first diagrams that are to be created in order for a system to be modeled.

The next UML behavior diagram to be explained is the activity diagram. The next figure shows an activity diagram example. The example models the Select Source use case on its multiple activities.

One can rightly infer that each one of the use cases that are described in a system multiple use case diagrams will most likely have an activity diagram detailing its functionality. It is also correct to interpret the activity diagram as the next abstraction layer from the use case diagram. It shows the procedural flow of control while processing an activity. It is probably one of the most flexible UML diagrams as it can be used to model high level business objectives and project requirements as well as low level implementation relationships.

Another advantage of this type of diagram is that it described not only the flow of activities in a sequential fashion, but also shows who is responsible for each of the activities.

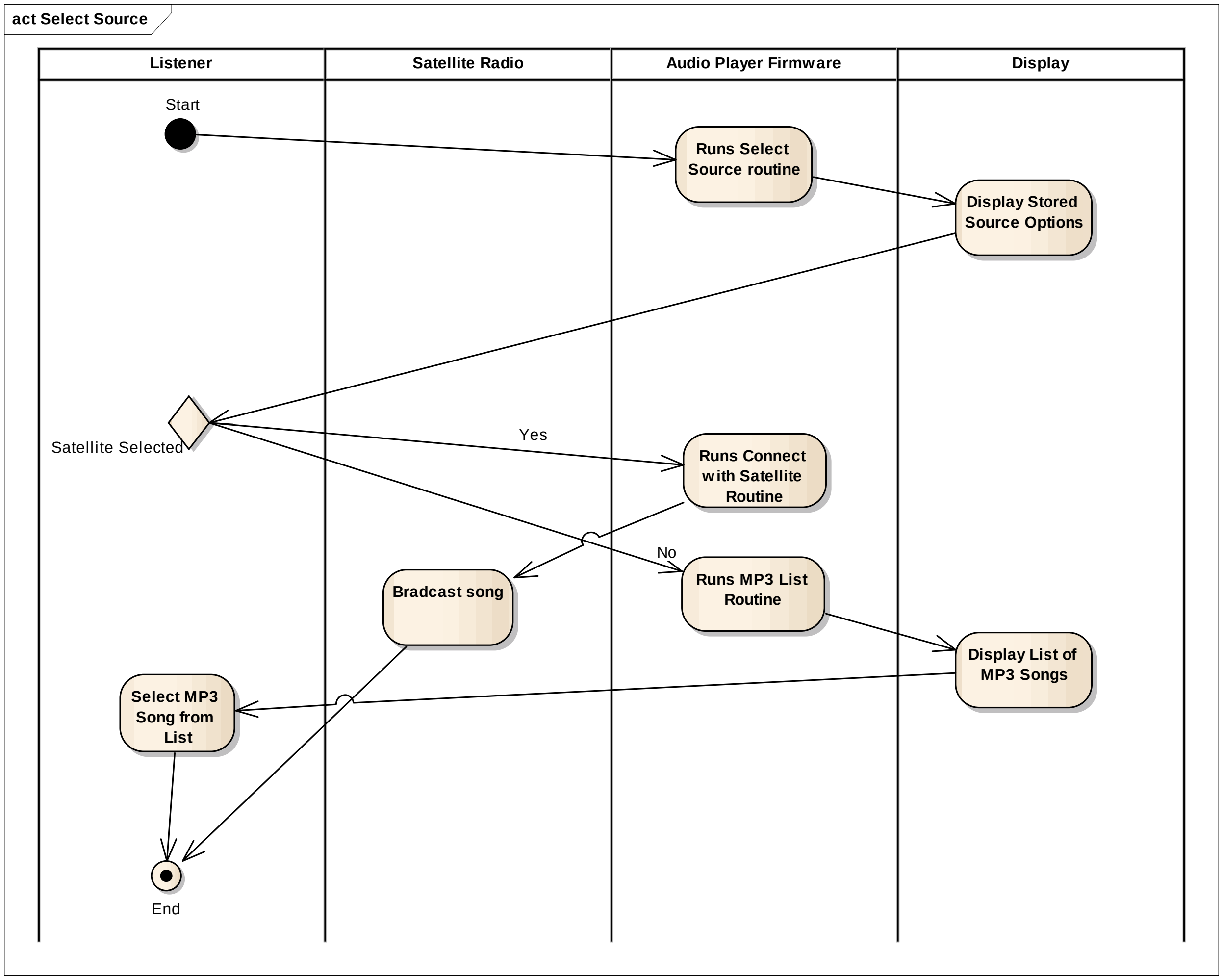
act Select Source
Listener
Satellite Radio
Audio Player Firmware
Display
Start
Runs Select Source routine
Display Stored Source Options
Satellite Selected
Yes
Runs Connect with Satellite Routine
No
Runs MP3 List Routine
Bradcast song
Display List of MP3 Songs
Select MP3 Song from List
End

Figure 4 – Select Source Activity Diagram

The diagram above shows how the Select Source use case activity would unfold. On this implementation of the activity diagram, it mixes high level information on who performs the selection and which actors are involved in the activity as well as low level information on which firmware routines are invoked and what shows up in the device display as a result of those routines executing.

Depending on the goal of the activity diagram, the information captured by the diagram may vary. One may elect to keep the relationships in the high level, if utilizing the diagram to collect business objectives and project requirements, or in the lower level only if describing system specifications or even design.

Structure Diagrams and the System Specification and Project Design

Previously in this article, it was mentioned that the UML structure diagrams emphasize the things that must be present in the system being modeled; or, “How” the system will implement “What” the system needs to do. One can also intuit that these types of diagrams would be great candidates to model the system design, the “Hows” of the system being built.

Since we are now talking about the system design, let’s introduce three UML diagrams that are extremely useful in the exercise of modeling the system design; class diagram, component diagram and deployment diagram.

The class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes or, the things that compose the system. Though a class is a construct of object orientation for Software programming, it is important to highlight that a class can very easily take other forms of constructs. It can represent a logical block of a system, or even a hardware component. The main idea that needs to be understood is that a class can be anything that can be self contained in a module of sorts, the building blocks of a system.

The class defines the attributes and behaviors that a system component will implement. The behavior is described by how it reacts to interfaces to other classes as well as other functions that it executes. The class diagram therefore shows how the system is being broken down into modules, as well as the static relationships between these modules.

The diagram below illustrates aggregation relationships between classes. The lighter aggregation indicates that the class "Account" uses AddressBook, but does not necessarily contain an instance of it. The strong, composite aggregations by the other connectors indicate ownership or containment of the source classes by the target classes, for example Contact and ContactGroup values will be contained in AddressBook.

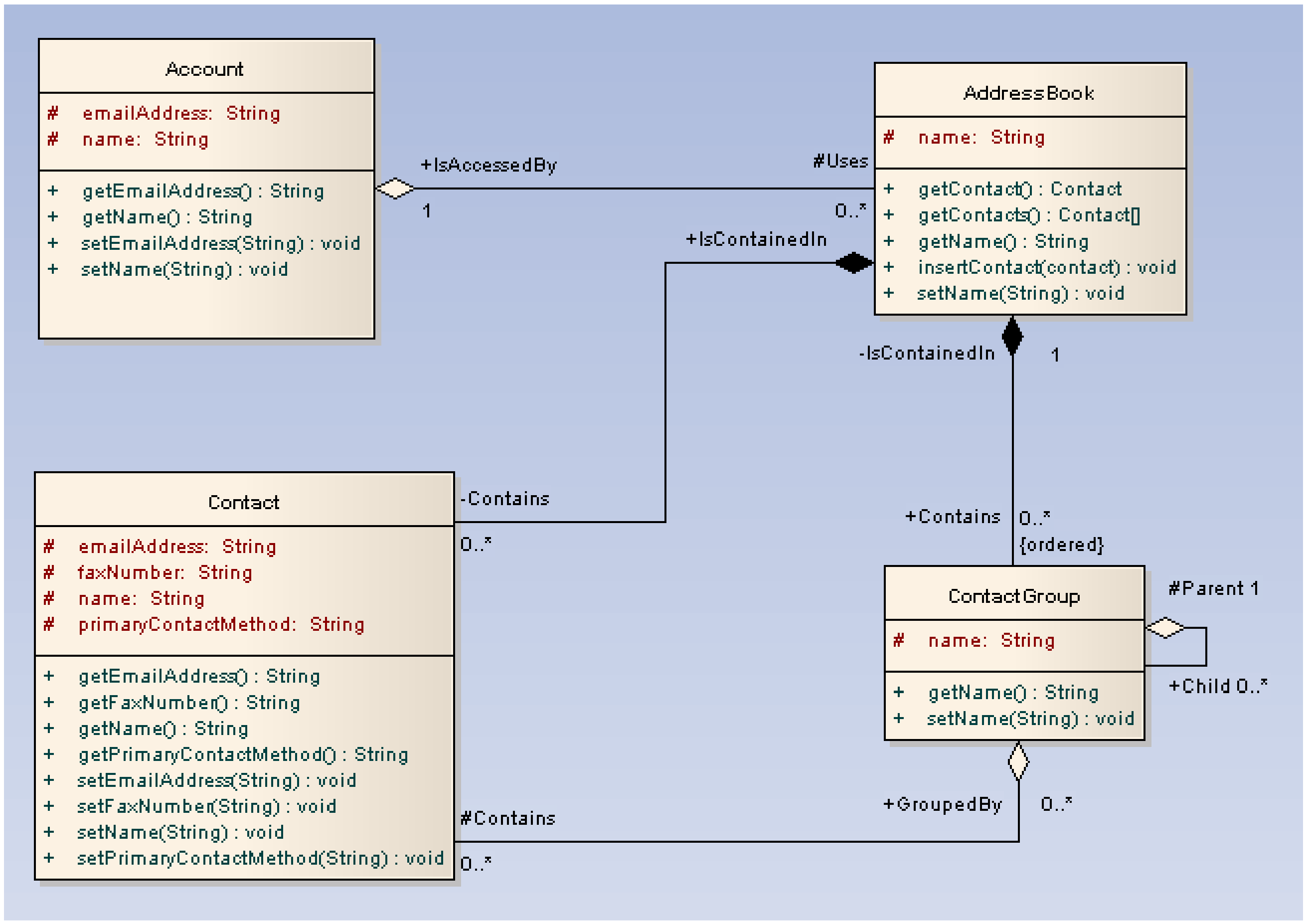

Figure 5 – Example of Class Diagram

The next immediate higher level of abstraction to the class diagram is what is called the component diagram. A component is implemented by one or more classes into larger logical building blocks. Components are great to compartmentalize classes into logical blocks of functionality. Again, the same comment that was made for classes is valid for components. Even though they started off as constructs to be used on modeling of Software systems, they can very well represent hardware logic blocks, such as controllers, instruments, fixtures, custom electronic boards, etc.

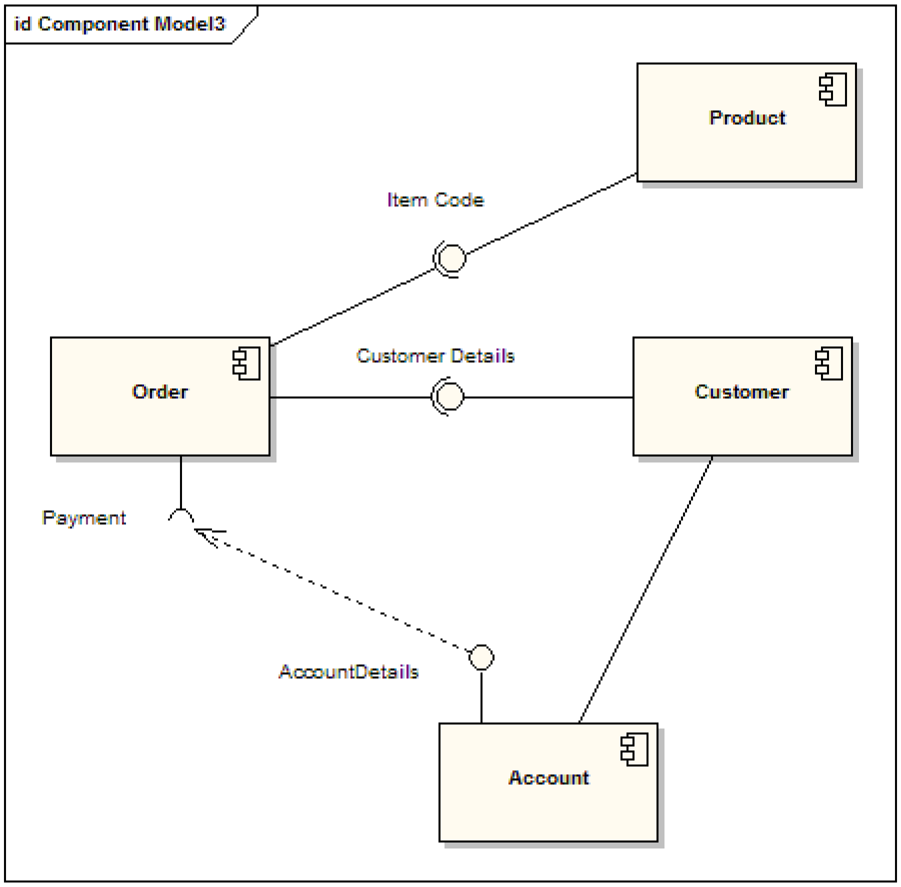

Figure 6 – Example of Component Diagram

The diagram above demonstrates some components and their inter-relationships. Assembly connectors "link" the provided interfaces supplied by "Product" and "Customer" to the required interfaces specified by "Order". A dependency relationship maps a customer's associated account details to the required interface; "Payment", indicated by "Order". Components define boundaries and are used to group elements into logical structures. The component modeled above sets a very clear boundary that only classes Order, Account, Customer and Product are part of it.

The last diagram that is work mention on this discussion is the so called deployment diagram. A deployment diagram models the run-time architecture of a system, or, in other words, the running test system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

Node

A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.

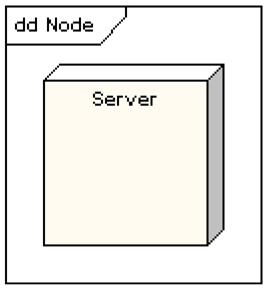

Figure 7 – Example of a Node

Node Instance

A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon. The following diagram shows a named instance of a computer. A node instance would be the equivalent of a class object in the object orientation software paradigm.

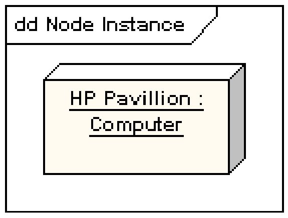

Figure 8 – Example of a Node Instance

Artifact

An artifact that may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals, etc.
An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon, as shown below.

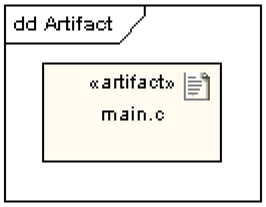

Figure 9 – Example of an Artifact

Association

In the context of a deployment diagram, an association represents a communication path between nodes. The following diagram shows a deployment diagram for a network, depicting network protocols as stereotypes, and multiplicities at the association ends.

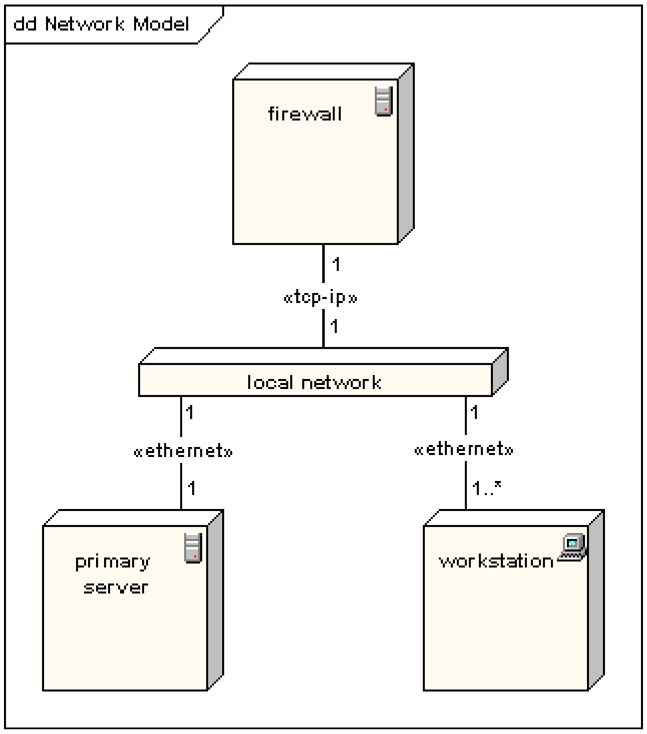

Figure 10 – Example of an Association

Node as Container

A node can contain other elements, such as components or artifacts. The following diagram shows a deployment diagram for part of an embedded system, depicting an executable artifact as being contained by the motherboard node.

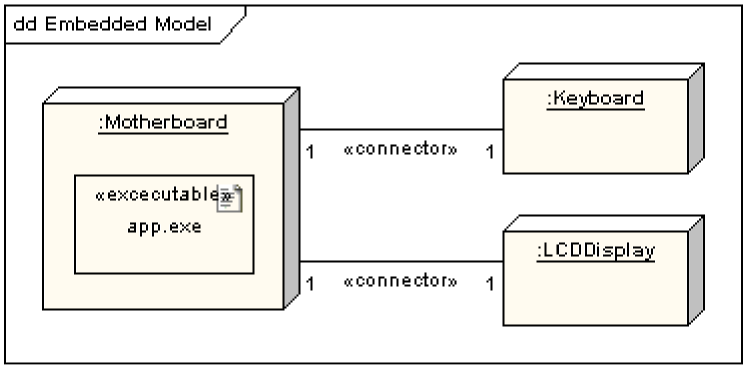

Figure 11 – Example of a Node as a Container

It is not difficult for one to see how these three UML diagrams, when used together can, not only fully represent the system specification and high level design, but also represent the full system design when the appropriate levels of abstraction are used.

Conclusion

This article presented a brief introduction to UML. It showed how UML's behavior diagrams are a good match to model test and measurements project objectives and requirements and how UML's structure diagrams are well suited to model system specification and design.

It presented the most useful diagrams for modeling project objectives and requirements, the use case diagram and the activity diagram. It also detailed three structure diagrams that are the main ones used for modeling of system specification and design; class diagram and component diagram.