# A SYSTEM OBJECT METHODOLOGY IMPROVES SIMULATION DEVELOPMENT

**Richard B. Wray**
**Lockheed Martin Tactical Defense Systems**
**Akron, Ohio 44315**
**1 December 1997**

## 1. METHODOLOGY BASIS

Third generation object oriented methodologies are key elements in modern simulation software design. More effective software development requires effective inputs from systems engineering, especially definition of simulation system requirements with an effective allocation to software elements, and a simulation system architecture which can be efficiently implemented by software engineering. The complexity of modern simulations require close cooperation in the methodologies in use by systems and software. Lockheed Martin Tactical Defense Systems (LMTDS) has complete initial results on an advanced implementation of key technologies for this object oriented system simulation in its Distributed Mission Training Technology (DMTT) initiative. Two key third generation approaches to software object modeling considered for this initiative are the Unified Modeling Language (UML) and the Fusion Methodology (FM).

For software engineering to take advantage of object development approaches such as UML or FM, systems engineering must follow compatible approaches to facilitate the software design of the simulation. The typical systems engineering functional approach causes problems in object oriented design of simulations due to the difficulties it creates in defining requirements in areas such as run-time operations and concurrent or sequential system behavior. There are also problems with the notations and process detailed elements. Use of incompatible methodologies creates a strong pressure on software engineering to independently redo the system requirements analysis using their own approach. Such a re-analysis invariably leads to slightly different results, which creates difficulties in system integration, test and customer acceptance. There is no commonly accepted object oriented system object methodology -- the purpose of this paper is to describe an object oriented system methodology based on a mixture of the precepts of both these software methodologies which better enables systems engineering to support effective startup of the software engineering process. Real-time developers can use these methods to model systems better than methods which inherently conflict.

The UML is promulgated by Rational Software Corporation, see their web site: *http://www.rational.com*. It adapts and extends the published works of Grady Booch, Jim Rumbaugh, Ivar Jacobson and dozens of others. It supports real-time systems particularly well by supporting classes, objects and many kinds of associations among them. Some of these such as use cases, aggregation, inheritance, and instantiation are useful to effective systems development. Use cases capture detailed descriptions of required system behavior and enable effective modeling of scenarios, including both timing and process synchronization behavior. Objects are defined to capture both data and the functions that operate on them.

The object model is key to the UML approach. *Class diagrams* are used to show the important sets of objects in the system and how they are related; boxes represent classes or objects. They are connected by lines representing some association between objects of these classes. There is a small diamond at one end to indicate ownership or aggregation. As the UML notes, “a *uses* association implies that a client class uses some facilities of a server class. An *aggregation* association means that one object is composed, at least in part, of another. An *inheritance* relationship means that one class inherits behavior and structure from the other; that is, one class is a specialization or extension of the other. An *instantiates* association exists between a parameterized class and the class that is created as a result of instantiation. Parameterized classes are specialized as to the type of information they work on. ” A subclass inherits data and behaviors of its superclass, and may extend them with new data and operations. Associations are

implemented in class diagrams by passing messages from one object to another, which abstracts the interface design details, enabling concentration on control dependencies among objects.

*Uses cases* are especially key to systems engineering since they describe broad behaviors the users need the simulation to offer, and which are key to successful software implementation. Use cases are the system object context diagrams, system behavior patterns and operating strategies which offer a high-level view of the system's functionality. This is critical for achieving understanding and agreement on simulation capabilities by users and engineers. Within a use case there are many threads of interaction that the system may execute. Each of these specific threads is called a *scenario*, i.e., a specific instance of the system behavior. For real-time systems, it is important to capture scenarios defining processes which can be concurrent and must be synchronized.

The FM is described in "Object Oriented Development: The Fusion Method" by Coleman et al. The typical systems engineering functional-based methodologies creates stumbling blocks in capturing requirements and generating designs for real-time interaction of objects. The FM is a method encompassing the analysis, design and implementation phases using object oriented approaches consistently. The key is defining the classes of objects, their relationships, operations to be performed and allowable sequences of operations during the analysis phase -- which is exactly the focus of systems engineering. The FM calls for definition of an object model and an interface model during analysis. The object model captures the concepts of the problem to be solved, their relationships and thus can be used to start defining the system requirements. A key element of the object model is the model of aggregation and inheritance for the classes developed. Another important element of the object model is the system object model created by excluding all classes and relationships in the environment. The system object model describes system structure, not behavior. While the FM uses a diamond notation to indicate relationships, for this paper all relationships are identified by notation on the line connecting classes or objects.

The interface model in the FM is used to describe system behavior in terms of events and the changes they cause. The environment is the set of entities the system communicates with. Events are the units of communication between the system and its environment and may consist of inputs or outputs. An event causes a change in the system state. An input and its effect are a system operation. The interface model consists of a life cycle model and an operations model.

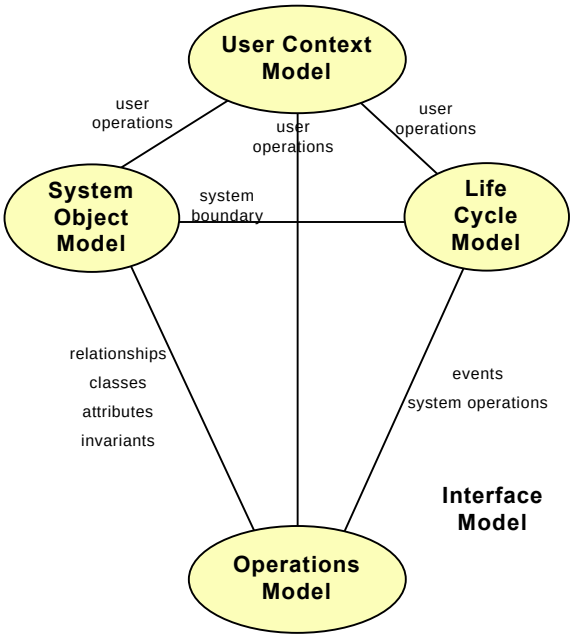


**Figure 1. Key Elements of the Object Oriented System Methodology**

The key elements of these approaches needed in the object oriented system methodology are consolidated (see Figure 1). A set of user context models are needed to fully describe how the users interact with the simulation system to accomplish their goals. A system object model is needed to describe the classes and behaviors in the system. The life cycle model is needed to describe the allowable operations and interactions over the life cycle of the system's operation. The operations model is needed to describe the individual system operations, which will lead to the software engineering data dictionary. The life cycle and operations models comprise the interface model which is an essential element needed early in the systems engineering process. The remainder of this paper describes how the UML and FM software methodology elements are assembled into a system methodology which complements and enables software simulation development.

The next section describes the user context models in the LMTDS DMTT initiative.

## 2. USER CONTEXT MODELS

There are two critical elements to the user context models: a simulation use context model and a system users operating context model. The simulation use context model identifies the training/exercise operations context which shows how the simulation system fits into a larger system picture. For an object oriented training methodology, this means the simulation use context model must be developed for the behavior of the distributed simulation in training operations and exercises. Systems engineering must describe the context of external elements and their interfaces to the system. The system users operating context model identifies the context for user interaction with the resulting system. This enables software engineering to accurately understand the scope of classes and behaviors which must be enabled by the simulation.

The first element in the system behavior context model describes the external entities and their interfaces to the system. The simulation use context model (see Figure 2) identifies the key operations inside and external to the simulation to be used for training in USAF missions. Key elements to the users are the missions in which training is needed and the exercises to conduct this training, as represented in the blue boxes. The user operational missions are converted into training and exercise mechanisms through the use of training objectives, resource plans, and exercise plans. Also outside the simulation being executed for training are the analysis tools to assess results, repositories to store results, and external libraries of objects and models to be drawn upon. If a simulation has the capability to add additional models and objects, then both internal configuration management and external configuration control are needed. These external entities establish the interfaces which must be defined for the simulation objects to be required. Within the simulation, classes are needed to enable logical instantiation simulating the scenario to be trained and its physical setup. Additional classes are needed as shown to enable a simulation to be run and results captured for later assessment or revision. The focus is on the external elements and how they interface to system elements.

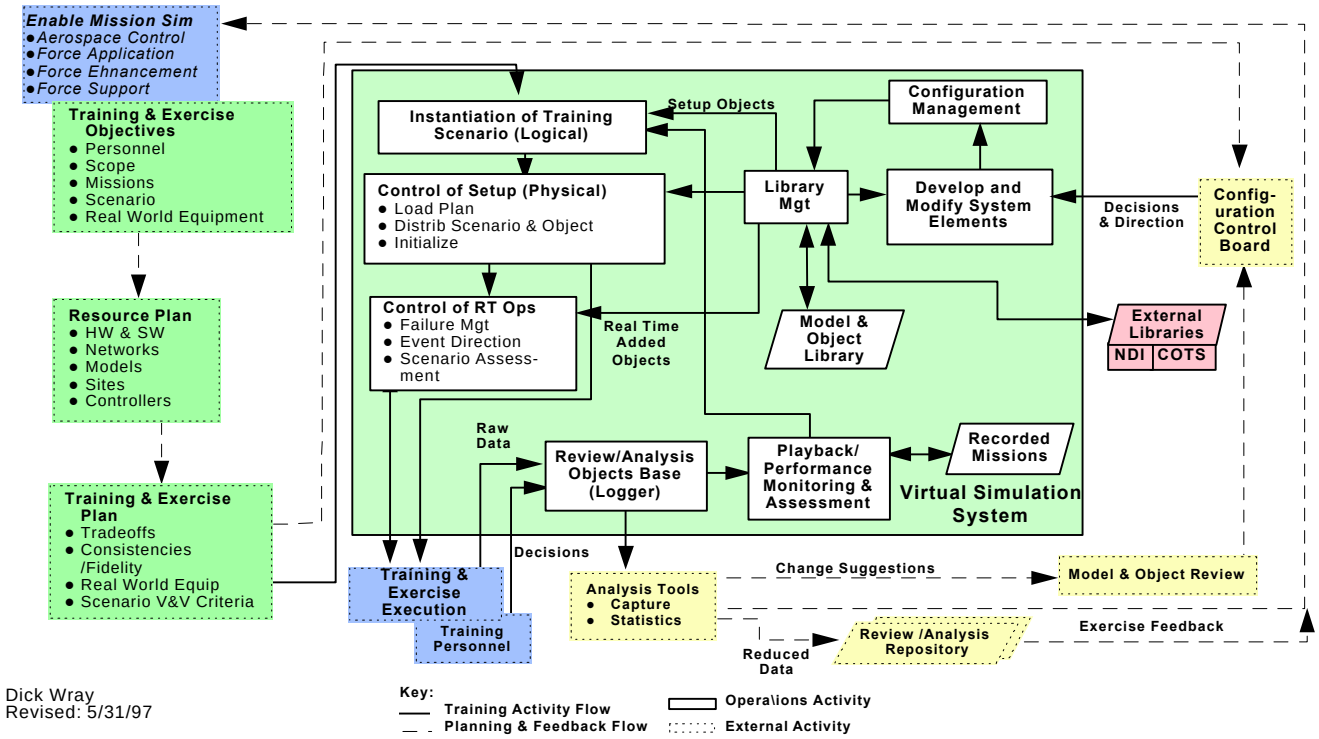


**Figure 2. Simulation Use Context Model**

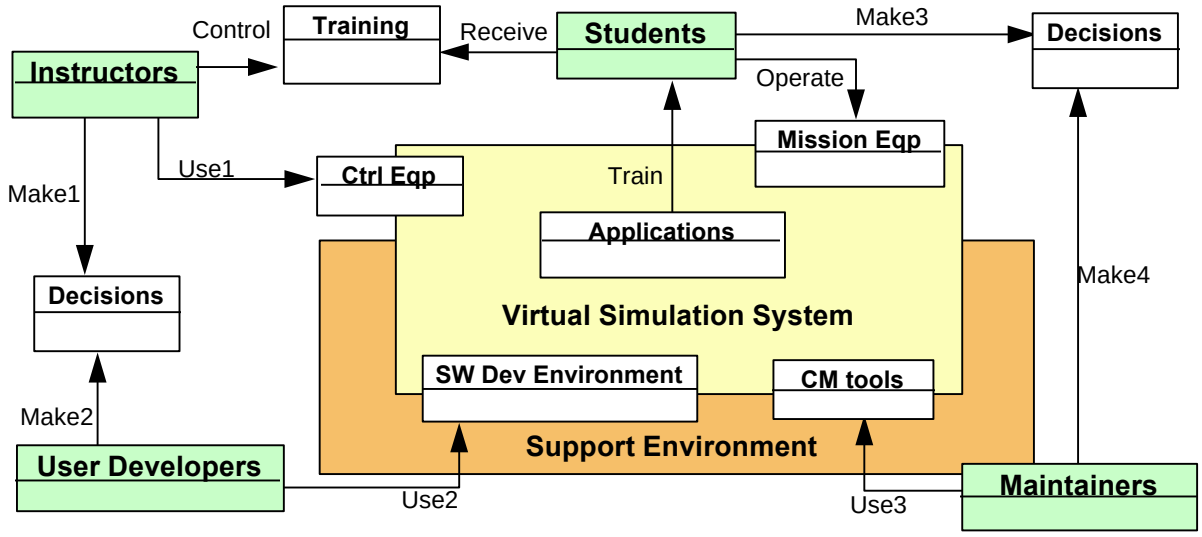


**Figure 3. System Users Operating Context Shows Relationships**

The next set of elements in the object oriented system methodology is definition of the user (classes) of the system. Then this methodology requires the definition of key system elements (classes) needed to enable users to interact with the system and each other. Finally, the relationships between users and the key system classes must be defined using a simplified operating context of the system for its users (see Figure 3).

Typically, four sets of users have been identified: instructors, students, user developers and maintainers. An effective system operating context model must show how each of these users interact with the system and the system classes through which each user class interacts. Some of the external classes which are important to using the system are the decisions made by people, and the training of the people since these have to be represented or otherwise handled by the simulation applications. Note that the class "decisions" is shown twice to minimize confusion from overlapping of lines. The association generally representing each interaction must be shown. For instance, there are several different kinds of "make" and "use" association which must be further described by systems engineering in the interface requirements.

Only those system classes which interface with user classes are shown within the system: the control equipment, mission equipment, applications, development environments and configuration management (CM) tools used for development and control. The fidelity of the mission equipment will depend on needs defined by the training requirements. System engineering defines the requirements for the classes in sufficient detail to enable software developers to perform the software engineering of the system and its behavior.

## 3. SYSTEM OBJECT MODEL

The system object model is used to organize the development of the software needed to prototype and implement the system. The system object model must identify the support environment if it is closely coupled with the system's operation, and the external elements with which it interacts.

The system object model from the DMTT initiative captures all system classes and relationships (see Figure 4). The emphasis is on identifying the key system classes for which behavior requirements are needed to effectively describe how the simulation interacts with its complete environment. Note that the previous set of users has been aggregated into the superclass "persons" and the software development environment and CM tools have been aggregated into the superclass "tools". This enables more complete systems object analysis of all people who might interact with the system, such as umpires and role players in a training exercise. It also enable more complete analysis of the tools needed, such as system simulation modeling tools. This more complete analysis thus would

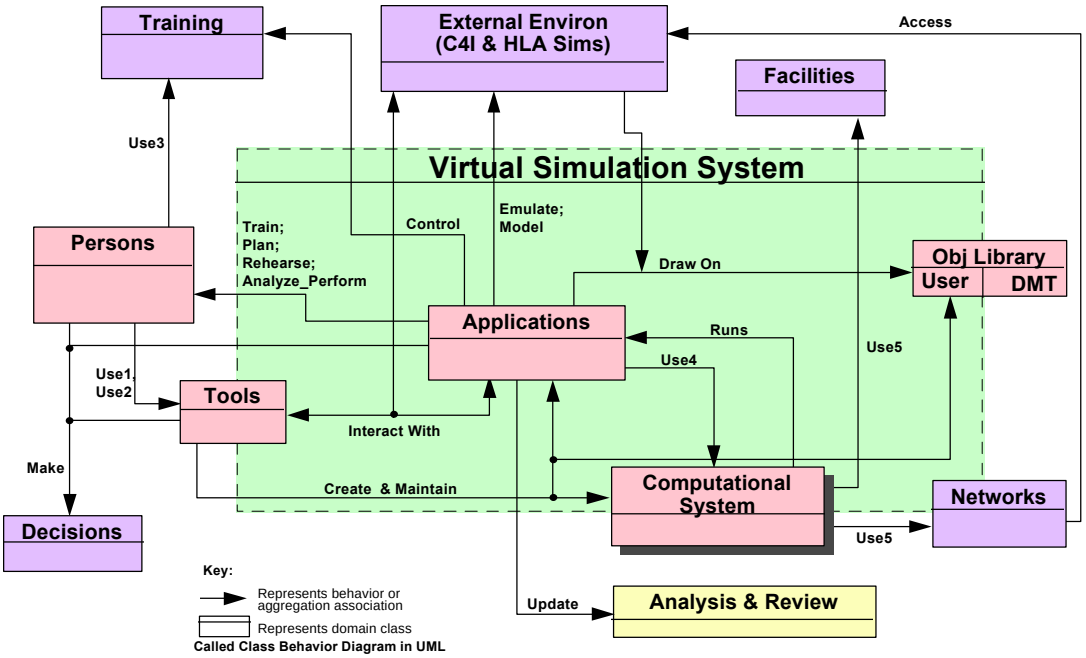


**Figure 4. Training/Exercise Simulation System Object Model**

need to be fed back into the user context models for completeness and consistency.

The system object model provides more high level detail into internal interfaces and behaviors of the system to facilitate early discussions with all stakeholders to insure that all required capabilities are captured in the user and system models. For instance, note that an "Analysis & Review" class has been added to this model. During discussions, this class and its object would be further defined to provide a better understanding early in the development process of its interactions with users and the simulation.

As all these models are further refined and expanded in engineering drawings, consistency must be maintained. Classes and their associations must be maintained in appropriate tools since their complexity will quickly overwhelm any manual or drawing system (e.g., MS Powerpoint™). Additional subclass diagrams can and should be used by systems engineering to ensure that their understanding is effectively captured in a form that can be passed along in a useful manner to software engineering. Thus, examples have been prepared (see Figure 5) illustrating the subclasses and interfaces internal to the class "Applications".

This enables discussions and accumulation of requirements for each of these. It also enables determination and agreement that all applications classes have been defined, interface relationships have been identified, and requirements allocation criteria are correctly understood and applied.

Associations need to be identified to describe the relationships across interfaces for system relationships or behavior in terms of events & the changes in state they cause. A system operation is an input event AND the effect it has. Examples have been prepared (see Figure 6).

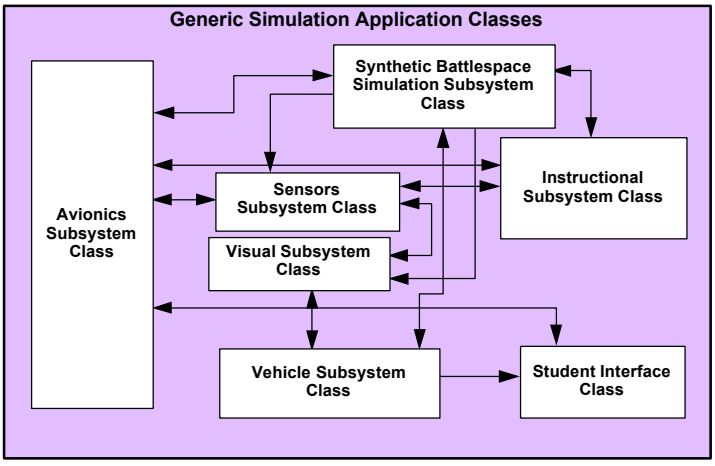


**Figure 5. Generic Application Object Model**

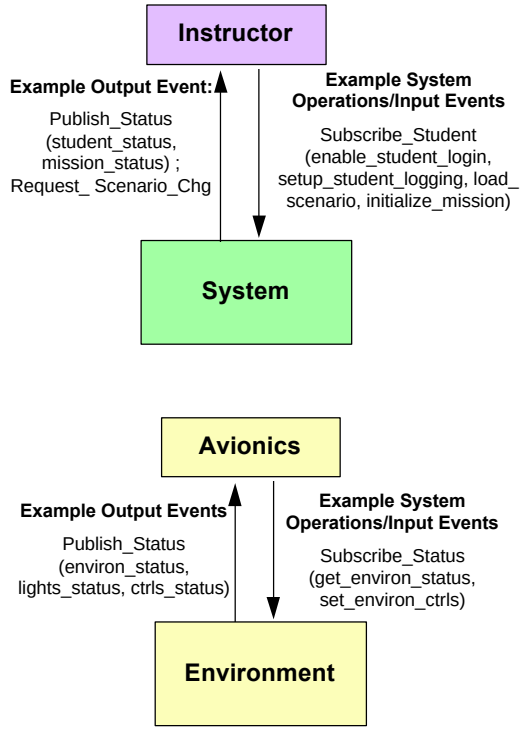


**Figure 6. Object Associations and Interface Behavior**

## 4. LIFE CYCLE MODEL AND SCHEMA

The life cycle model is one of the two key elements of the interface model. It identifies and describes the allowable operations and sequences of interactions over the system's life cycle which the software must implement.

The life cycle model of a system is defined (see Figure 7). This captures the behaviors of the system over its entire life cycle to ensure all operations are effectively developed and executed. An example of a schema for the life cycle model is shown below. In this example, the DMTT is implemented by the optional operation "develop", followed by "plan_execution", then "setup", then "execute", then "stop" and followed by the optional operation "review_&_analyze". The "develop" operation provides output "objects", or "models", or "scenarios". Each of these operations is expanded on following lines in more detail until a primitive executable operation is reached.

DMTT = [**Develop**. (#Objects | #Models | #Scenarios). **Test**. (#Validated_Objects | #Validated_Models | #Validated_Scenarios)]. **Plan_Execution**. (#Instantiation. #Mission). **Setup. Execute**|| (Checkpoint | Observe | Tag | Direct)*. **Stop**. #Record. [**Review_&_Analyze**]

where:

- **Name (DMTT) is a name for a life cycle element, = is implemented by , followed by a life cycle expression**
- **The life cycle expression can consist of operations or more life cycle expressions:**

**Develop** is an operation
Description: Perform knowledge acquisition, engineering analysis, design, implementation, and V&V of models and object for inclusion in the Object_Library.
Reads: —
Changes: Object_Library (or that + Model_Lib, Scenario_Lib, Mission_Lib)
Sends: —
Assumes: —
Result: the verified and validated models are added to the library
**#Models** is a product or output
**Plan_Execution** is an expression
Plan_Execution = Ident_Scenario. Ident_Classes. Ident_Resources.
and Ident_Scenario is an operation

and where: an element is a process, = is 'implemented by', . is separator 'and' or 'then', [ ] is 'optional', # is 'output' or 'produce', || is 'while', ( ) is 'doing', | is 'or', e* is zero or more repetitions of expression e, and e+ is one or more repetitions of e.

## 5. OPERATIONS MODEL AND SCHEMA

The operations model is the second of the two key elements of the interface model. It describes the specific operations named in the life cycle model, and identifies key elements needed in the schema.

**Life Cycle Model Schema**

**Name =** life-cycle-expression

Identifies allowable operations and sequences of interactions over the system's life cycle

**Figure 7. Life Cycle Model Definition**

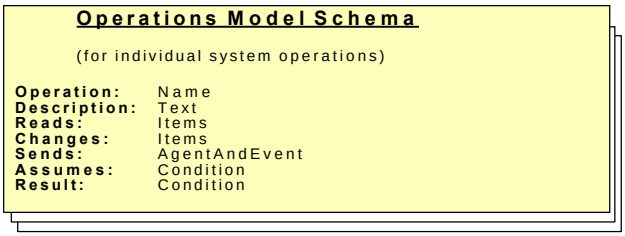


**Figure 8. Operations Model Definition**

The operations model of a system is defined (see Figure 8). This captures An example of an operations schema is shown below. In this example, descriptions, items read, items changed, items sent, assumptions, and resulting conditions are defined for the DMTT. This is analogous to the software data dictionary, except it addresses system operations. Three system operations are shown here: distribute_sim_objects, ident_scenario, and insert_new_object. Discussions of the system operations clarify the operations required and thus described for all program personnel. Discussions can also identify operations disallowed by implication since they are not shown in the operations model schema. For instance, the potential operation "delete_object" might be disallowed implicitly if not shown in the schema at all (depending on rules setup for using the schema), or explicitly under defined conditions if shown and noted under the description (as shown below) that object deletion can only be accomplished during certain phases of the simulation. When systems engineering establishes all operations (with the assistance of software), then unclear requirements for executing operations are reduced.

- **Distribute_Sim_Objects**
  Description: Load models, objects, and data structures into hosts and networks following a Load_Plan.
  Reads: Load_Plan, Model_Object_Library
  Changes: Distributed_Computing_Environment
  Sends:
  Assumes: a valid Load_Plan exists
  Result: the Distributed_Computing_Environment is ready for Initialization
- **Ident_Scenario**

Description: Identify the Area_Of_Operations, real-world equipment, and Forces that will be used for a particular simulation instantiation.
  Reads: Model_Object_Library
  Changes: Instantiation
  Sends:
  Assumes: —
  Result: a new Instantiation object is created with a consistent set of classes and methods
- **Insert_Sim_Object**
  Description: Instantiate a new object within an existing simulation execution.
  Reads: Controller
  Changes: Execution, Record
  Sends:
  Assumes: an Execution exists; Controller has proper authority
  Result: a new simulation object has been instantiated
- **Delete_Object**
  Description: Delete an existing object within an existing simulation execution only during scenario instantiation.
  Reads: Controller
  Changes: Execution, Record
  Sends:
  Assumes: an Execution exists; Controller has proper authority
  Result: an existing simulation object has been removed from the instantiation

NOTE: in order to ensure system consistency throughout an exercise, control events such as the creation of a crater on an airfield or the removal of an aircraft from the simulation are handled using Insert_Sim_Object to insert an event onto the simulation event cue. There is no direct means of removing or modifying an object in a simulation execution.

## 6. AGGREGATION AND INHERITANCE SUPPLEMENT SYSTEM MODELS

Models of the aggregation and inheritance of the classes are needed to supplement the system models to improve common understanding of the classes, their object and association required by the user and defined by systems engineering. A virtual simulation (VS) system as established in the Lockheed Martin DMTT consists of five classes: persons, tools, object library, applications, and the computational system. Both the inheritance trees (subclasses) and the aggregation trees (parts-of) for these classes and their objects have been identified in the DMTT (see Figure 9). In the computational subsystem, the parts-of this subsystem established by DMTT are the software such as operating system (OS) services and hardware. The subclasses are the software applications which must inherit services from the computational subsystem to operate. Common development of this model by systems and software engineering improve it utility to both domains.

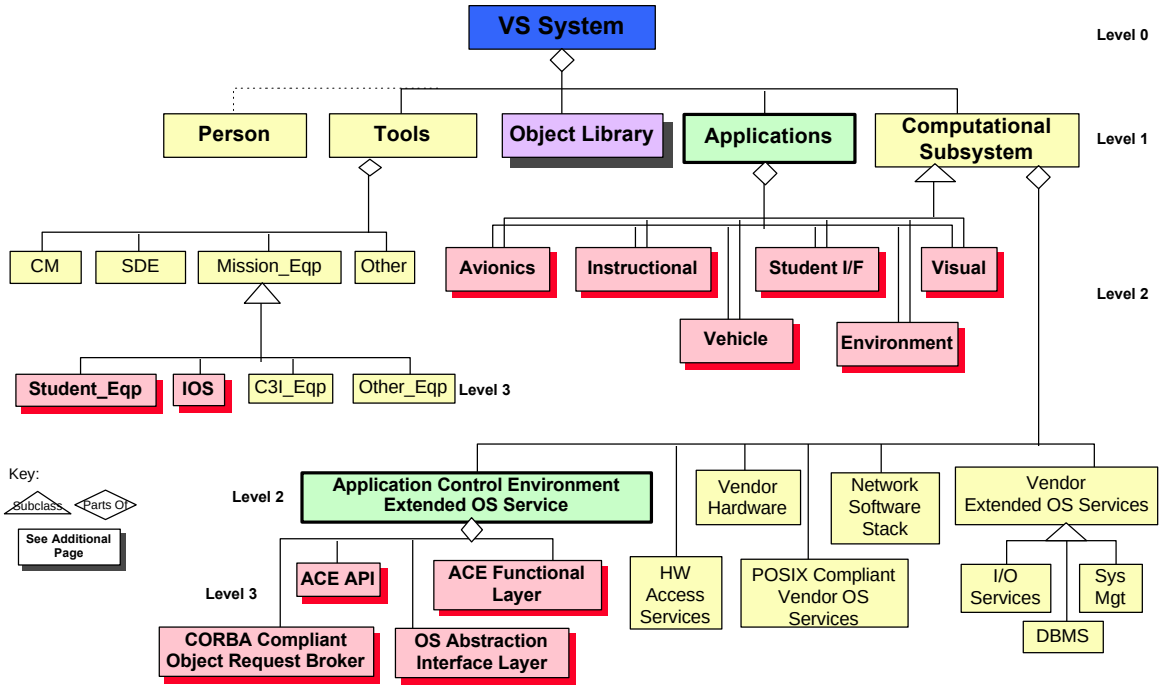


**Figure 9. Models of Aggregation & Inheritance in System Modeling**

## 7. SUMMARY

There are four key models needed for effective system modeling: User Context Model, System Object Model, Life Cycle Model, and Operations Model. These are supplemented by models of aggregation and inheritance for classes in the system under design. Their implementation follow the precepts of the Unified Modeling Language and the Fusion Method.

The advantages of object oriented approaches typically include data abstraction, compatibility, flexibility, reusability, extensibility, and ease of maintenance. These are as valid for systems engineering as for software engineering. If software engineering wants to take advantage of object development, then systems engineering should also follow this approach to facilitate the software design of the simulation. The functional approach will cause problems in object oriented design of the simulation due to the difficulties it creates in defining requirements in areas such as run-time operations and concurrent or sequential system behavior. There are also problems with the notations and process detailed elements. Use of incompatible methodologies creates a strong pressure on software engineering to independently redo the system requirements analysis using their own approach. Such a re-analysis invariably leads to slightly different results, which creates difficulties in system integration, test and customer acceptance.

The object oriented systems methodology presented in this paper establishes an initial framework which should be used by systems engineering to organize their analysis efforts. That would significantly improve the transition of requirements to software engineering and the design of the subsequent simulation.