Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

AN OBJECT-ORIENTED FRAMEWORK FOR SIMULATING MULTI-ECHELON INVENTORY SYSTEMS

Manuel D. Rossetti
Mehmet Miman
Vijith Varghese
Yisha Xiang

Industrial Engineering Department
4207 Bell Engineering Center
University of Arkansas
Fayetteville, AR 72701, U.S.A

ABSTRACT

In this paper, we discuss the design and use of an object-oriented framework for simulating multi-echelon inventory systems. We present a context for how the framework can be used through its application on two examples. In addition, we describe the design by examining the major conceptual artifacts within the object-oriented model. The framework is built on a Java Simulation Library (JSL) and permits easy modeling and execution of simulation models. The results and discussion indicate the flexibility and power of modeling with the framework. In addition, we summarize our future research efforts to model complex supply chains.

1 INTRODUCTION

In general, a supply chain can be considered as a network of facilities and distribution options that operate to obtain raw materials, transform these materials to finished products, distribute these finished products to the customers depending upon their demand and repair the failed products. Engels and Lee (2000) also consider a supply chain as “a network of entities that starts with the suppliers’ supplier and ends with the customers’ customer for the production and delivery of goods and services”. In this paper, we present an object-oriented simulation framework for simulating a supply chain with an arborescent tree structure. These types of supply chains form a general class of inventory systems called *multi-echelon inventory systems*. Figure 1 illustrates the basic structure of a multi-echelon inventory system. This research is part of a larger effort to develop an object-oriented framework for simulating supply chains and especially supply chains that are found within the military’s new sense and respond logistics paradigm. The concept of sense and respond logistics has envisioned the need to rapidly design responsive demand and support networks that can anticipate, predict, and coordinate actions at the strategic, operational, and tactical levels (Rossetti 2005). A framework to model *multi-echelon inventory systems*, with options and flexibility to model each point in the network with unique behavioral characteristics, is an essential component in developing simulations of more responsive supply networks.

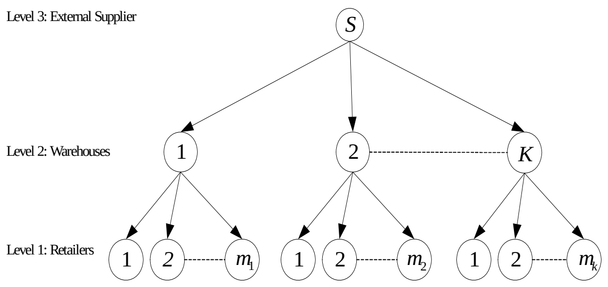


Figure 1 Multi-Echelon Inventory System

Analytical solutions using mathematical tools such as probability theory and optimization can be used to obtain the approximate performance of multi-echelon inventory systems. For research and references along those lines, we refer the interested reader to Al-Rifai and Rossetti (2006), Zipkin (2000), and Muckstadt (2005). Unfortunately, the analytical modeling of complex supply chains is limited to the underlying mathematical assumptions that make the mathematics tractable. For the performance modeling of complex supply chains, simulation turns out to be an excellent method for the evaluation of most of the systems where the relationships among the components are very complex. Dong (2001) considered simulation as a better technology for designing supply chain systems due to the system variation and interdependencies. Ingalls (1998) concluded that simulation is the best method to analyze supply chain systems where the key driver is variance. On the other hand, simulating complex supply chain networks has also some disadvantages . The major problem for using simulation for complex supply chain analysis is that most of the simulation models are specific to a particular problem and have a limited use (Rossetti and Chan 2003). The two major problems associated with simulation models are 1) they may take a long time to develop and 2) they are very specific and have limited reuse (Swaminathan *et al.* 1998).

Most of the Commercial Off-The-Shelf (COTS) simulation packages do not easily simulate general purpose supply chains. This is not meant to imply that they cannot simulate supply chains, but rather that their modeling constructs have been focused on other types of modeling domains. For many simulation packages, their primary purpose has been the simulation of manufacturing and material handling systems. Because the performance analysis of supply chain networks is essential within industry, many commercial *simulators* are available, which will simulate particular supply chain scenarios. Supply Chain Operational Performance Evaluator (SCOPE), SIMLOX, the Logistics Simulation and Analysis Model (LogSAM), are three such simulators within the domain of spare part inventory systems. The SCOPE simulation model was developed for the review and evaluation of readiness based sparing models for the Navy and for the review of efficient management and transportation policies for the Defense Logistics Agency. SCOPE models the entire logistics support for spare parts from the retail echelon to the whole sale echelon of weapon systems by monitoring the failures of the spare parts through four levels of indenture (Culosi 2001). SIMLOX is a Systecon's discrete event Monte-Carlo simulation model developed for simulating operational, maintenance logistics, and their interactions for any technical systems (Systecon 2001). SIMLOX models features such as full system and sub-system modeling capability without indenture constraints, data driven, modeling of lateral resupply and various repair philosophies (repair versus replacement). LogSAM modeling architecture simulates critical aircraft generation functions based on known and verifiable aircraft-specific reliability and maintainability statistics and can be used to test alternative supply and maintenance processes under a wide variety of resource-constrained situations.

The purpose of our research effort is to bridge the gap between commercial general purpose simulation packages and specific supply chain simulators by developing open source frameworks for performing simulations. Booch *et al.* (1999), define a software design framework as "an architectural pattern that provides an extensible template for applications within a domain." A framework provides a set of abstract and concrete classes that can be extended via sub-classing or used directly to solve a particular problem within a particular domain. This paper discusses a software design framework for object-oriented simulations. Such a framework can ease the work of researchers, educators, and practitioners. An object-oriented simulation framework can provide a better understanding of key abstractions within simulation modeling and can provide a blueprint for the development of object-oriented simulation libraries. See for example the work of Pratt et al. (1991). Our research is not only about simulating supply chains but also about finding and describing the key architectural patterns that are found in this domain.

In order to do this we are building on the work of Rossetti and Chan (2003), and Rossetti and Thomas (2005). Rossetti and Chan (2003) built a supply chain modeling framework which contains logistics elements like facility, warehouse, product etc. This Supply Chain Simulation Framework (SCSF) facilitates the dynamic analysis of multi-echelon supply chain systems. The SCSF consists of 29 classes representing the various elements within a supply chain network. The most important of all relationships in the SCSF is the Relationship Network. A Relationship Network is defined as a complex system of interconnected network nodes that exchange material and information in order to provide material, products or services to the end-users (Rossetti and Chan 2003). Using the knowledge gained from the SCSF, Rossetti and Thomas (2005) built a modeling architecture for spare parts supply chain networks, especially those related to multi-indentured weapon systems. This framework expands the notion of a facility to a behavioral agent-based plug-in architecture that allows customization of facilities through user-defined behaviors.

In our modeling, we have identified a key layer of abstraction for the modeling of supply chains, which we term the *inventory layer*. Currently, the other layers in our overall framework include a *facility layer* and a *transport layer*. In this paper, we discuss the inventory layer. We do this by describing the object-oriented constructs within the inventory layer and by illustrating their use in simulating multi-echelon inventory systems. Our object-oriented framework is built upon a Java Simulation Library (JSL), see Rossetti *et al.* (2000). The JSL is a simulation library for Java. The JSL's current version has packages that support random number generation, statistical collection, basic reporting, and discrete-event simulation modeling. The development of a simulation model is based on sub-classing the ModelElement class that provides the primary recurring actions within a simulation and event scheduling and handling. The user adds developed model elements to an instance of Model and then executes the simulation. The JSL is divided into Java packages (calendar, examples, modeling, spatial, observers, and utilities). The modeling package is further divided into packages that facilitate the modeling of processes, resources, queues, transporters, etc. In this paper, we describe the basic architecture of the *jsl.modeling.elements.supplychain.inventory* package.

In what follows, we first give a basic overview of the structure and functionality of the inventory package. We then describe a simple example of an (r,Q) inventory model to illustrate the primary capabilities of the architecture. Finally, we illustrate how easy it is to develop models of multi-echelon inventory systems through a larger but still simple example. We wrap up with a summary of our efforts and present ideas for future research.

2 INVENTORY PACKAGE

In this section, we present the inventory layer through two examples (1) a simple (r,Q) continuous review inventory model and (2) a multi-item multi-echelon inventory system. The inventory package currently consists of 18 classes and 8 interfaces representing various elements within inventory modeling. A complete discussion of all of the implementation details of these classes is beyond the scope of this paper. Our intention here is to provide enough detail so that the reader can make conceptual modeling with the inventory package more concrete.

In order to illustrate our design, we give a brief description of six important classes:

- ItemType
- Demand
- DemandGenerator
- BackLogPolicyAbstract
- Inventory
- InventoryPolicyAbstract.

An ItemType represents or describes the items or products in the inventory system. The Demand class represents a request for inventory and provides the status of the request. An instance of Demand records the item type associated with the request, the customer of the request, the provider of the inventory items, the shipper of the items, the amount of the request, and other customer requirements associated with the request, such as whether backlogging or partial filling is permitted. The DemandGenerator class creates instances of Demand and acts as a customer that generates the demand by providing details of the customer requirements. The Inventory class represents units of items that can be requested and keeps track of the amount of inventory on-hand, backlogged, on order, lost, etc. An instance of the Inventory class represents the state of the inventory at any given time. The Inventory class provides methods for requesting inventory and for filling demands for the inventory. Every Inventory class is associated with an inventory policy. The InventoryPolicyAbstract is an abstract base class that allows for the encapsulation of rules to control the associated inventory; it is a rule, policy, or strategy that governs the reordering behavior for inventory. The inventory policy determines when to order and how much to order. A number of different inventory policies have been implemented within the inventory package. Also associated with an Inventory class is a backlog policy. An abstract base class, BackLogPolicyAbstract, represents the different rules or behaviors that can be used to backlog demands for inventory and to fill backlogs associated with inventory. The relationships between these 5 classes are illustrated in Figure 2. In the following example, we illustrate the use of these classes and the way in which they interact.

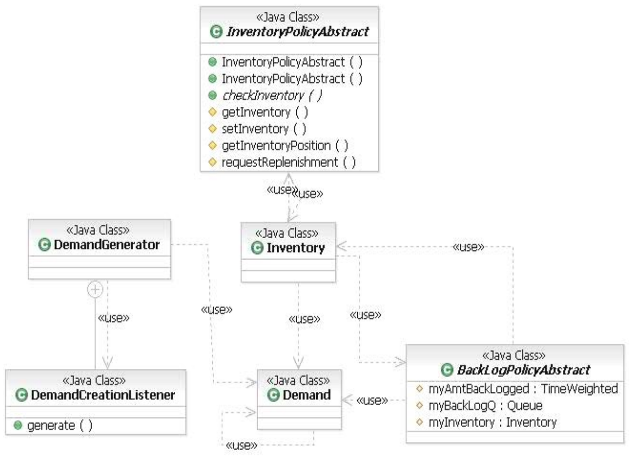


Figure 2 Class Diagram of Primary Inventory Classes

2.1 Simulating a simple (r,Q) Inventory Model with the Inventory Package

The flow chart in Figure 3 illustrates the basic logic associated with an (r, Q) inventory model with backlogging. When a demand occurs, the amount demanded is determined, and then the system checks the availability of stock. If the stock on-hand is sufficient for the order, the demand is filled and the quantity on-hand is decremented. On the other hand, if the stock on-hand is not sufficient to fill the order, the entire order is backordered. The backorders are accumulated in a queue and they will be filled on a FIFO basis after the arrival of replenishment order. The inventory position is checked each time after a regular customer demand and the occurrence of a backorder. When the inventory position falls under the reorder point, a replenishment order is placed. The replenishment order will take a certain time to arrive and subsequently backorders are filled and the on-hand inventory is incremented.

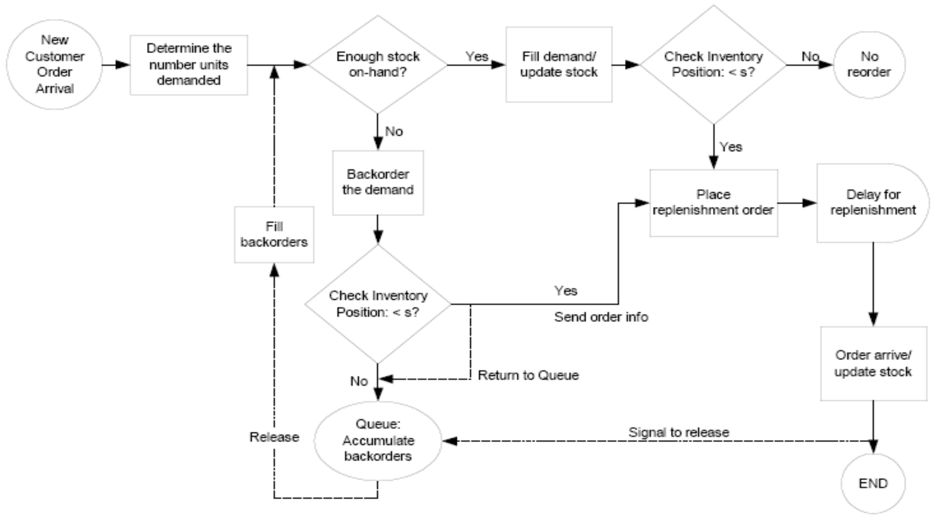

New Customer Order Arrival
Determine the number units demanded
Enough stock on-hand?
Yes
Fill demand/ update stock
Check Inventory Position: < s?
No
No reorder
No
Yes
Backorder the demand
Place replenishment order
Delay for replenishment
Fill backorders
Check Inventory Position: < s?
Yes
Send order info
Order arrive/ update stock
No
Return to Queue
Queue: Accumulate backorders
Release
Signal to release
END

Figure 3. Flowchart of the (r, Q) Inventory Logic

Using the inventory package, we developed and tested various scenarios of the (r, Q) inventory model (Figure 3). Table 1 presents the basic parameters of each of the model runs and the corresponding analytical (A) and simulation (S) results. The simulations were run for 30 replications of 3360 days with a warm up period of 360 days. In this simple example, the demands occur according to a Poisson process with rate of 3.6 units per day. The lead time for replenishment orders is a constant of (0.5) days.

Table 1: Basic (r,Q) Simulation Results

(Q=2)	**Avg. Stockout Frequency**		**Avg. Inventory**	
r	A	S	A	S
-1	0.92	0.92±0.001	0.09	0.08±0.001
0	0.69	0.69±0.001	0.40	0.40±0.002
1	0.41	0.40±0.002	1.00	0.99±0.004
2	0.19	0.19±0.002	1.81	1.80±0.005
3	0.08	0.07±0.001	2.73	2.73±0.006

The code for creating a simulation and running an experiment is quite simple as illustrated in Exhibit 1.

```
Model m = Model.createModel();
DistributionIfc tbd = new Exponential(1.0/3.6);
DistributionIfc lt = new Constant(0.5);
int rpt = -1;
int qty = 2;
int level = 0;
RQInventoryModel rq = new RQInventoryModel(m,
tbd, lt, rpt, qty, level);
Experiment e = new Experiment(m);
e.setNumberOfReplications(30);
e.setLengthOfReplication(3360.0);
e.setLengthOfWarmUp(360.0);
e.runAll();
```

Exhibit 1: Creating and Running a Model

As can be seen in Exhibit 1, we need to create only an instance of the class RQInventoryModel. This class uses the previously described 6 classes. Exhibit 2 presents the primary functionality of the constructor of the RQInventoryModel. The first line defines an ItemType and represents the items that can be demanded. The next 2 lines create a LeadTimeReplenishmentProvider. A LeadTimeReplenishmentProvider acts as an inventory provider when an inventory requires a replenishment. A LeadTimeReplenishmentProvider is a special case of Inventory in which an infinite amount of inventory can be supplied after a time delay.

```
ItemType myItemType = new ItemType(this, "Type-
A");
LeadTimeReplenishmentProvider supplier = new
LeadTimeReplenishmentProvider(this);
supplier.addLeadTime(myItemType,leadTime);
DefaultBackLogPolicy bpolicy = new DefaultBacLog-
Policy(this);
myInventoryPolicy = new RQInventoryPolicy(this,
reorderPt, reorderQty);
myInventory = new Inventory(this, myItemType,
supplier, myInventoryPolicy, supplier, ini-
tialLevel, bpolicy, false);
myDemandGenerator = new DemandGenerator(this,
timebtwDemand, timebtwDemand);
myDemandGenera-
tor.setDemandRequirements(myItemType, true,
false, myInventory);
```

Exhibit 2: Constructor for RQInventoryModel

The next two lines define the policies for the instance of the Inventory class. The DefaultBackLogPolicy is a sub-class of the abstract base class BackLogPolicyAbstract, and the RQInventoryPolicy is a sub-class of the InventoryPolicyAbstract. The next line creates an instance of the Inventory class that uses the supplied policies, sets the supplier, and gives the initial amount on hand. Finally, the last two lines define the DemandGenerator and sets its requirements. In this case, it is generating demands for the given item type, allows backlogging, does not permit partial filling, and sends its requests to the previously defined inventory.

In this model, events are initiated when demands are created. This is implemented by the generateAction() method encapsulated in DemandGenerator class as shown in Exhibit 3. The method first checks if the provider can provide the inventory by calling getInventory() method. If a reference to a valid inventory is returned then a demand representing the request is created by requestInventory-Method(). If the demand is not null, the fillInventory() method is invoked to fill the demand.

```
protected void generateAction(EventGenerator gen-
erator, JSLEvent event){
    if (myInventoryProvider != null){
        // get the amount of the demand
        int amt = 0;
        if (myAmtRV == null)
            amt = 1;
        else
            amt = (int)myAmtRV.getValue();
            InventoryIfc i = myInventoryProvider.
    getInventory(myItemType, this);
        if (i != null){

            Demand d =
    i.requestInventory(getItemType(), this, amt);

        if (d != null)
            i.fillInventory(d);
        }
    }
}
```

Exhibit 3: Generating Demands and Requesting Inventory

In this logic, demands are created by the Inventory class. Demand represents a “contract” at the current time to potentially fill a request. The demand object has information concerning how much can be filled at the current time, whether the request will be backlogged, etc. The sender of the request (in this case a DemandGenerator) can decide to have the demand filled or not, depending on the status of the request. In this simple case, the demand is then given back to the inventory class via the fillInventory() method and asked to be filled at the current simulation time.

The fillinventory() method of the Inventory class deals with the request for the demand initially according to its request status, such as FILL_ALL, BACKLOG_PARTIAL, BAKLOG_NO_PARTIAL, LOST_PARTIAL and LOST_ALL, which is set when the demand is requested via the requestInventory() method. After checking the demand’s request status, the demand is processed via an internal handleInitialRequest() method. If the demand cannot be filled immediately, the Inventory class uses the instance of DefaultBacklogPolicy, to handle the backlogging of the demand (if permitted). In the case where the demand can be filled, it is necessary to invoke the inventory control policy. The inventory evaluation event is managed through the inventory policy specified, RQInventoryPolicy in our case, whose mechanism is provided by the InventoryPolicyAbstract class. For example, associated with the above case, handleInitialRequest() triggers the checkInventory() method of the supplied RQInventoryPolicy. Exhibit 4 presents a code-snippet from the Inventory class and the RQInventory-Policy class. From this code, it is easy to see that the Inventory class delegates the policy checking behavior to its supplied policy, making it easy to change policies before and during a simulation run.

```
    // within Inventory class
    protected final void checkInventory(){
        myInventoryPolicy.checkInventory();
    }

    // within RQInventoryPolicy
    public void checkInventory() {
        int ip = getInventoryPosition();
        if (ip <= myReorderPoint){
            requestReplenishment(myReorderQty);
        }
    }
```

Exhibit 4: Invoking the Inventory Policy

In summary, the RQInventoryModel uses an RQInventory-Policy to control the inventory. It uses a DemandGenerator to generate arrivals, similar to standard CREATE modules in languages like Arena. In this RQInventoryModel, backlogging is permitted while no partial filling is allowed. The backlog is processed by the instance of DefaultBackLog-Policy where the discipline for the backlog queue is FIFO. An object of the Inventory class is used to represent the inventory and keep track of its states. RQInventoryModel also uses an instance of the LeadTimeReplenishmentProvider class to provide the replenishment for inventory. The Lead-TimeReplenishmentProvider allows time-based shipping to replenish the inventory. The sequence diagram of the implementation is provided in Figure 4.

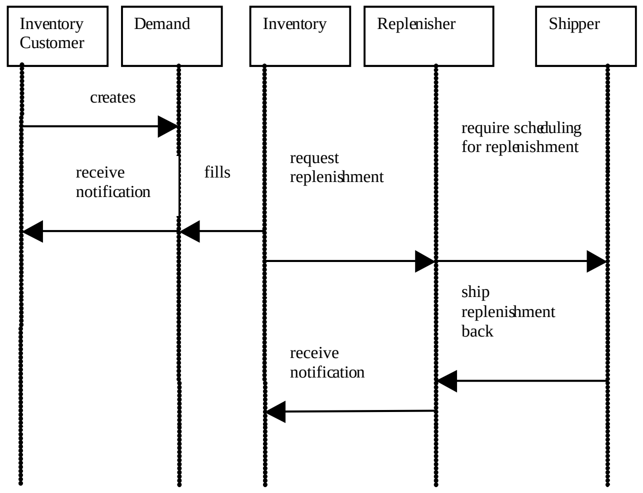


Figure 4 Sequence Diagram Between Major Objects

While much of the underlying logic has been omitted from this paper, we can see that the inventory package has many capabilities for easily modeling inventory systems. We have implemented and tested other inventory policies and the structure of the framework allows for easily plugging-in behavior to make supply chains. In the next example, we illustrate how the inventory package can be used to simulate a multi-echelon inventory system, simply by defining and "hooking" up the objects within the system.

2.2 Simulating a Multi Echelon System with the Inventory Package

The functionality of important classes in the Inventory Package has been demonstrated in detail through the a (r,Q) Inventory Model. This section illustrates the use of the inventory package to modeling a typical multi item multi echelon inventory system.

Figure 5 illustrates a three echelon supply chain handling demands for 4 items. There are four retailers: Retailer A, B, C and D, having demands for Items 1, 2, 3 and 4, at echelon 1, where the demand arrives according to a Poisson distribution. The retailers follow a (r,Q) inventory policy and have a warehouse as their inventory provider, which also follows a (r,Q) inventory policy. The lead time for each item shipped from the warehouse to the retailers is constant. The warehouse is replenished by a factory which manufactures in response to the order with a production delay which is also constant.

We begin by discussing the key set of classes and interfaces that facilitate the model building for multi item multi echelon inventory systems.

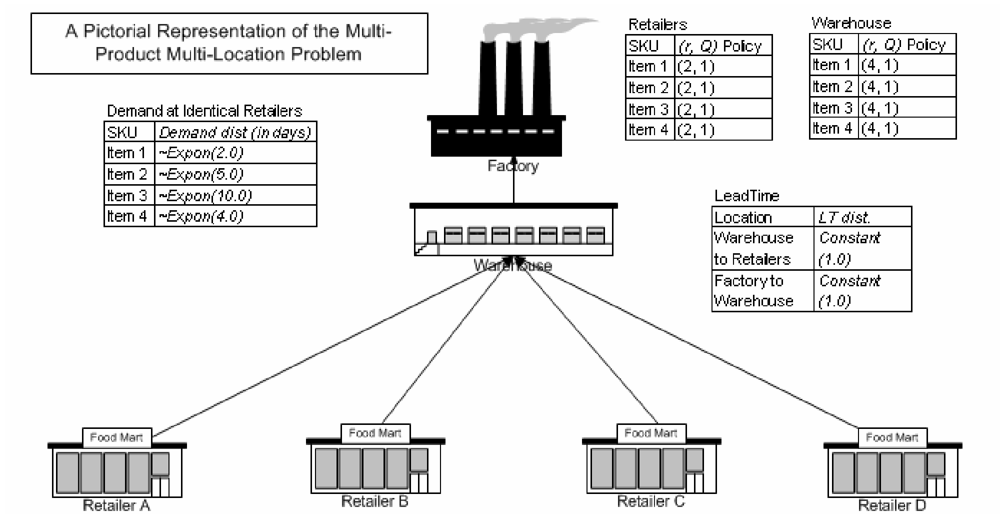
- InventoryIfc
- InventoryCustomerIfc
- InventoryProviderIfc
- InventoryShipperIfc
- InventoryHoldingPoint
- LeadTimeReplenishmentProvider
- NoDelayShipper
- TimeBasedItemTypeShipper

InventoryCustomerIfc declares all the operations pertaining to a customer in an inventory system and the InventoryProviderIfc describes the behavior of a provider of inventory. An inventory provider can be asked if it provides inventory that meets a customer's requirements by means of the providesInventory() method and then the getInventory() method can be used to actually get access to the inventory at a particular provider's location. However this transaction is permitted only if the customer and provider permits an agreeable transaction (for example: both may permit backlogging or not or both may permit partial filling or not etc.). This is achieved by means of permitReplenishmentBackLogging and permitReplenishmentPartialFilling methods for customers.

InventoryShipperIfc facilitates shipping the items from the provider to the customer. In order to facilitate the replenishment of shipments, there is an abstract base class InventoryShipperAbstract that can be used to model this behavior. For example, two typical shippers are: NoDelayShipper where a replenishment is achieved instantly and TimeBasedItemTypeShipper to cause a delay with which a replenishment is made based on the type of item being shipped.

The InventoryHoldingPoint class, encapsulates all the features of holding inventory at a particular location. An inventory holding point has all the behaviors of inventory through an implementation of the InventoryIfc interface; it can act as a customer by implementing InventoryCustomerIfc and can act as a provider by implementing the InventoryProviderIfc interface. Thus, an inventory holding point acts as both a customer (to other inventory holding points) and a provider (to other inventory holding points) within the inventory system. Figure 6 illustrates the attributes and the operations of the IHP and the interfaces it implements. The LeadTimeReplenishmentProvider is quite similar to the InventoryHoldingPoint except that it has itself as its own inventory provider. LeadTimeReplenishmentProvider has a TimeBasedItemTypeShipper object embedded within it and it causes a delay in replenishment to its customer as per the specified the distribution. This allows the top of the multi-echelon network to be modeled with a location that has an infinite supply of items that can be requested with a time delay.

With these basic classes/interfaces defined, we can now model the example. The 4 SKUs within the system are first declared and their associated DemandGenerator objects created as discussed in the previous section. The former is achieved through the object ItemType which takes the ID of the SKU as an argument (illustrated pictorially in Figure 5). The Demand object associated with each ItemType object is the "entity" moving within the system. The events related to these entities causes the simulation to run.



A Pictorial Representation of the Multi-Product Multi-Location Problem

Demand at Identical Retailers

SKU	*Demand dist (in days)*
Item 1	*~Expon(2.0)*
Item 2	*~Expon(5.0)*
Item 3	*~Expon(10.0)*
Item 4	*~Expon(4.0)*

Retailers

SKU	*(r, Q)* Policy
Item 1	(2, 1)
Item 2	(2, 1)
Item 3	(2, 1)
Item 4	(2, 1)

Warehouse

SKU	*(r, Q)* Policy
Item 1	(4, 1)
Item 2	(4, 1)
Item 3	(4, 1)
Item 4	(4, 1)

LeadTime

Location	*LT dist.*
Warehouse to Retailers	*Constant (1.0)*
Factory to Warehouse	*Constant (1.0)*

Figure 5. Multi Product Multi Location Problem

All the retailers have the attributes of an inventory system as well as an inventory provider (retailer is an inventory provider to the customer demand: DemandGenerator object) and an inventory customer (retailer is an inventory customer of the warehouse: InventoryHoldingPoint object).

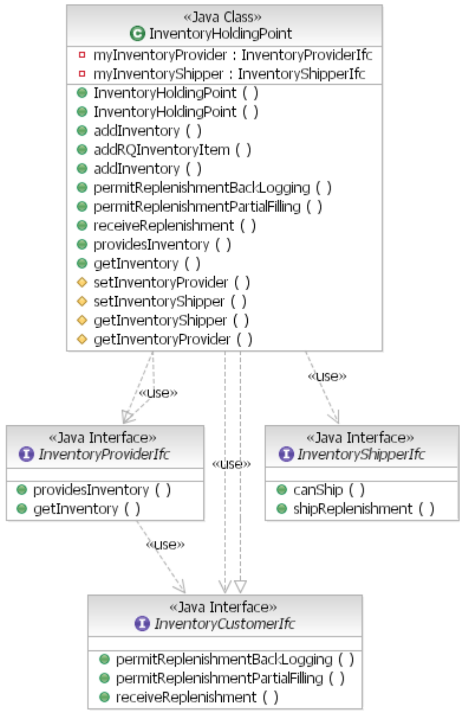


Figure 6 InventoryHoldingPoint

Since the InventoryHoldingPoint class encapsulates all the above three roles, we create the retailer A, B, C and D as instances of the InventoryHoldingPoint class. Similarly, we declare the warehouse also as an instance of the InventoryHoldingPoint class, because it has the attributes of an inventory system as well as an inventory provider (to the retailers) and an inventory customer (to the factory, e.g. LeadTimeReplenishmentProvider). In the example, the factory meets the orders of the warehouse with a production delay and we declare it as a LeadTimeReplenishmentProvider. The retailer A, retailer B, retailer C, retailer D, warehouse and factory are the major model elements within the simulation model and they are declared within the code as in Exhibit 5.

```
ItemType type1 = new ItemType(m, "Type-1");
ItemType type2 = new ItemType(m, "Type-2");
ItemType type3 = new ItemType(m, "Type-3");
ItemType type4 = new ItemType(m, "Type-4");

LeadTimeReplenishmentProvider supplyFactory = new
LeadTimeReplenishmentProvider(m);
InventoryHoldingPoint warehouse= new Inventory-
HoldingPoint(m, supplyFactory, warehouseShipper,
"W");
InventoryHoldingPoint retailerA = new Inventory-
HoldingPoint(m, warehouse, retailerShipper, "R-
A");
InventoryHoldingPoint retailerB = new Inventory-
HoldingPoint(m, warehouse, retailerShipper, "R-
B");
InventoryHoldingPoint retailerC = new Inventory-
HoldingPoint(m, warehouse, retailerShipper, "R-
C");
InventoryHoldingPoint retailerD = new Inventory-
HoldingPoint(m, warehouse, retailerShipper, "R-
D");
```

Exhibit 5: Declaring the Elements of the System

The IHP receives requests from its customers. For example, in this problem RetailerA is an IHP and its customer is the DemandGenerator which generates demand say for Item-1. When the demand is generated, it identifies its inventory provider and requests inventory (it passes a reference of the Inventory object) and if both parties meet the requirements (backlogging, partial filling etc.), then the customer triggers the fillInventory() method in the IHP (inherited from the InventoryIfc, Figure 10) and the IHP triggers a shipment to the customer by the shipReplenishment() method of the shipper class assigned to it. The retailer A is assigned a NoDelayShipper and when shipReplenishment() is invoked the customer receives its replenishment. In this case the replenishment is achieved with no delay. However, if the TimeBasedItemTypeShipper is assigned to the retailer, the replenishment can be easily scheduled with a delay.

Each time a demand is fulfilled or replenishment from demands arriving to an IHP, the inventory is checked according to its inventory policy. The InventoryPolicy, if it finds that a replenishment from the provider has to be made, it then requests its provider for replenishment. Eventually, its provider schedules the shipment to the IHP and when the shipment arrives also, the inventory policy checks its inventory.

Inventory at each IHP is designated by item type and added to each IHP accordingly. For the example, we used an (r,Q) inventory policy for each item at each location and we allow backlogging but do not allow partial filling. This is achieved by the addRQInventoryItem() method in the InventoryHoldingPoint class. Exhibit 6 has a code snippet showing how the method is invoked to add inventory at retailer B.

```
   retailerB.addRQInventoryItem(type1,2,2,1);
   retailerB.addRQInventoryItem(type2,2,2,1);
   retailerB.addRQInventoryItem(type3,2,2,1);
   retailerB.addRQInventoryItem(type4,2,2,1);
```

Exhibit 6 :Adding Inventory to an IHP

Once all the Inventory objects of each item type are added at the retailers and the warehouse, we create objects that will facilitate shipping the items from factory to warehouse, warehouse to retailer, and retailer to the customer. In order to schedule shipments of replenishment, the inventory package has implemented shipper classes that can schedule delay for replenishment. The shipment from retailer to the customer is to be achieved instantly and an instance of the NoDelayShipper class is created for this purpose as in Exhibit 7 and is assigned to the retailers when the retailer objects are created (Exhibit 5).

```
// The retailer uses a NoDelayShipper to ship to
its customers
NoDelayShipper  retailerShipper  =  new  NoDelay-
Shipper(m);
```

Exhibit 7: NoDelayShipper Class for Demand Fulfillment

The warehouse ships its items to the retailers facilitated through an instance of the TimeBasedItemTypeShipper class which is created as in Exhibit 8. The distribution that governs the lead time is added to the shipper and is assigned to the warehouse when the warehouse object is created (Exhibit 5).

```
TimeBasedItemTypeShipper  warehouseShipper  =  new
TimeBasedItemTypeShipper(m);
warehouseShipper.addLeadTime(type1,lt1);
warehouseShipper.addLeadTime(type2,lt2);
warehouseShipper.addLeadTime(type3,lt3);
warehouseShipper.addLeadTime(type4,lt4);
```

Exhibit 8: TimeBasedItemTypeShipper generation

The factory is modeled as an instance of LeadTimeReplenishmentProvider and the TimeBasedItemTypeShipper which is embedded within it allows the assignment of the lead times of each item type via the addLeadTime() method (Exhibit 9).

```
// within the LeadTimeReplenishmentProvider
  DistributionIfc pt1 = new Constant(1.0);
  supplyFactory.addLeadTime(type1,pt1);
```

Exhibit 9: Lead Time Assignments

Once the shippers are assigned the model is ready to be run. We set a replication length of 10 years and a warm up period of an year and the experiment is replicated 30 times. The results are summarized for the item tye1 for the purpose of illustration in the Table 2. (The half widths for all the responses were 0.00 for all the performance measures indicating at least two significant digits).

Table2: MIME Results for Item-1

Type-1 Results	Stock-out Freq.	Avg. OnHand Inv.	Avg. OnOrder Inv.	Avg. Back-logged
WareHouse	0.05	3.02	2.00	0.02
Retailer-A	0.01	2.50	0.50	0.00
Retailer-B	0.01	2.50	0.51	0.00
Retailer-C	0.01	2.50	0.51	0.00
Retailer-D	0.01	2.50	0.50	0.00

We also modeled the above problem in Arena in order to compare the performance measures of stock-out frequencies as well as average inventories and backorders at each echelon (Table 2). The results were the same to within statistical error. Although the model here is small, there are no physical limitations (except for memory) that limit the size of the model (e.g. number of echelons, retailer, warehouse, inventory items, allocation of inventory to each IHP, etc.

3 SUMMARY

In this paper we have introduced and illustrated the use of the *jsl.modeling.elements.supplychain.inventory* package, which has been developed as a part of an object-oriented framework for simulating supply chains, particularly the ones within the military's new sense and response logistics where the structure of the supply chain can be enormously complex and dynamic.

The inventory package, currently consisting of 18 classes and 8 interfaces and is built upon a Java Simulation Library. By defining and "hooking" up the objects a variety of inventory systems can be simulated by redefining the input parameters. For example, an instance of "Demand" can be used to deal with a variety of complex systems in which demand can be backlogged, lost or partially lost along with partially filling options or not. Similarly, a Demand Generator can describe a variety of cases where demand arises from any distribution governing the time between demand. In addition, the InventoryPolicyAbstract can be used to implement a variety of inventory policies. Currently the (r,Q) and (s,S) inventory policies in continuous as well as periodic schema are built into the inventory package.

In this paper we did not provide a complete discussion of all of the implementation details of all classes, instead we provide enough detail on important classes along with their important behaviors in order to illustrate their use as well as functionality through two examples. Hence, the reader can make conceptual modeling with the "Inventory Package" developed for these cases concrete. In addition, it should now be clear that a variety of complex systems can be modeled where at each echelon, at each inventory holding point, as well as for each item type a variety of different inventory policies, backlogging policies, and demand shipment options can be used. In addition, because the framework is object-oriented and built on Java, the modeler can use all the power of the object-oriented modeling and Java to develop additional models and behaviors.

Currently, the aggregation of performance at each echelon to provide the overall assessment of the system evaluation, is under development. It should be also noted that the package described here constitutes only the "inventory layer" of the supply chain framework and on-going research is being directed to combine this layer with the "facility layer" and the "transport layer" to simulate and evaluate entire supply chains from the demand arising from a customer point to being produced by supplier, and transported back through multiple echelons to the customer. The facility layer will handle complex order processing and the transport layer will handle the complexities of truck load and less-than-truck load shipping.

In addition to these modeling efforts, work is in progress to provide database persistence to the supply chain models developed within the framework, from which the parameters for the model, such as each item's distributions, lead time, BOM, etc. can be supplied through to the simulation model easily along with overall cost components for the entire supply chain. Finally, all of the these research activities are meant to be used for the evaluation of sense and respond logistics systems and to facilitate their optimization.

ACKNOWLEDGMENTS

This material is based upon work supported by the Air Force Office of Sponsored Research and the Air Force Research Laboratory. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force.

REFERENCES

Al-Rifai, M. H. and M. D. Rossetti. 2006. An Efficient Heuristic Optimization Algorithm for a Two-Echelon (R, Q) Inventory System. Accepted for publication in the *International Journal of Production Economics*.

Booch, G., J. Rumbaugh, and I. Jacobson., 1999. *The Unified Modeling Language User Guide*. Addison-Wesley.

Culosi, S. J. 2001. A simulation for evaluating the operational readiness of the supply chain. MacLean, VA.

Dong, M. 2001. Process modeling, performance analysis and configuration simulation in integrated supply chain network design. Doctoral Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA.

Engels, G. and L. Groenewegen. 2000. Object-Oriented Modeling: A roadmap, In *Proceedings of the Conference on the Future of Software Engineering*, ed. A. Finkelstein, 105-116. Limerick, Ireland.

Ingalls, R. G. 1998. The value of simulation in modeling supply chains. In *Proceedings of the 1998 Winter Simulation Conference*, ed. D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, 1371-1375. Piscataway, New Jersey: Institute of Electrical and Electronic Engineers.

Pratt, D. B, P. A. Farrington, C. B. Basnet, H. C. Bhuskute, M. Kamath, and J. H. Mize. 1991. A framework for highly reusable simulation modeling: separating physical, information, and control elements. In *Simulation Symposium, Proceedings of the 24th Annual*, 254 –261. New Orleans, LA, USA.

Muckstadt, J. A. 2005. *Analysis and Algorithms for Service Parts Supply Chains*. New York: Springer.

Rossetti, M. D., B. Aylor, R. Jacoby, A. Prorock, and A. White. 2000. Simfone′: An object-oriented simulation framework, In *Proceedings of the 2000 Winter Simulation Conference*, ed. J. Joines, R. Barton, P. Fishwick, and K. Kang, ACM/SIGSIM, ASA, IEEE/CS, IEEE/SMCS, IIE, INFORMS/CS, NIST and SCS, 1855-1864.

Rossetti, M. D., and H. T. Chan. 2003. A Prototype Object-Oriented Supply Chain Simulation Framework. In

Proceedings of the 2003 Winter Simulation Conference, ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, ACM/SIGSIM, ASA, IEEE/CS, IEEE/SMCS, IIE, INFORMS/CS, NIST and SCS.

Rossetti, M. D. 2005. Modeling and Simulation Based Framework for Sense and Respond Logistics Concepts. Report AFSOR Tasks, Department of Industrial Engineering, University of Arkansas, Fayetteville, AR.

Rossetti, M. D. and S. Thomas. 2005. Object-Oriented Multi-Indenture Multi-Echelon Spare Parts Supply Chain Simulation Model. Accepted for publication in the *International Journal for Modeling and Simulation*.

Systecon. 2001. SIMLOXv2 logistics simulation from Systecon [Homepage of Systecon], [Online]. Available: <http://www.systecon.co.uk/simlox.htm> [accessed on 2002] .

Swaminathan, J.M. 1998. Modeling supply chain dynamics: A multi-agent approach. *Decision Sciences* 29 (3): 607-632.

Zipkin, P. H. 2000. *Foundations of Inventory Management*. McGraw-Hill Companions, Inc.

AUTHOR BIOGRAPHIES

MANUEL D. ROSSETTI, Ph. D., P. E. is an Associate Professor in the Industrial Engineering Department at the University of Arkansas. He received his Ph.D. in Industrial and Systems Engineering from The Ohio State University. Dr. Rossetti has published over thirty-five journal and conference articles in the areas of transportation, manufacturing, health care and simulation and he has obtained over $1.5 million dollars in extra-mural research funding. His research interests include the design, analysis, and optimization of manufacturing, health care, and transportation systems using stochastic modeling, computer simulation, and artificial intelligence techniques. He was selected as a Lilly Teaching Fellow in 1997/98 and has been nominated three times for outstanding teaching awards. He is currently serving as Departmental ABET Coordinator. He serves as an Associate Editor for the International Journal of Modeling and Simulation and is active in IIE, INFORMS, and ASEE. He served as co-editor for the WSC 2004 conference. His email and web addresses are <rossetti@uark.edu> and <www.uark.edu/~rossetti>.

MEHMET MIMAN is a PhD student in the Department of Industrial Engineering at the University of Arkansas. Prior to coming to the University of Arkansas, he received his M. I.E. from North Carolina State University, and B.S. in Industrial Engineering from Bilkent University. He served in NCSU as a Teaching Assistant for Statistical Quality Control for two years and taught Production Planning and Control Course in Atilim University. Currently he is a Research Assistant in the University of Arkansas and a member of IIE and INFORMS. His primary research interests are in Reliability and Supply Chain Management.

VIJITH M. VARGHESE, M.S.I.E. is a Research Assistant and a Ph.D. student in the Department of Industrial Engineering at the University of Arkansas. He received his M.S. in Industrial Engineering from the University of Arkansas and B.Tech. in Mechanical Engineering from the Mahatma Gandhi University, India. His areas of interest include computer simulation, demand modeling and forecasting, and the design and analysis of supply chain systems.

YISHA XIANG is a Research Assistant and a Ph.D. student in the Department of Industrial Engineering at the University of Arkansas. She received her B.S. in Industrial Engineering from Nanjing University of Aero.&Astro., China. Her area of interest is in repairable system modeling.