LA-UR -90-3692

LA-UR--90-3692

DE91 001850

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE OBJECT-ORIENTED INVENTORY CLASSES: COMPARISON OF IMPLEMENTATIONS IN KEE (a frame-oriented expert system shell) and CLOS (the Common Lisp Object System)

AUTHOR(S) Richard R. Silbar and H. W. Egdorf

SUBMITTED TO 1991 Western Simulation Multiconference, January 23-24, 1991, Anaheim, CA

DISCLAIMER

NOV 0 5 1990

MASTER

Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO. 836 R4
ST. NO. 2629 5/81

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Object-Oriented Inventory Classes: Comparison of Implementations in KEE and CLOS

Richard R. Silbar and H. W. Egdorf

Los Alamos National Laboratory, University of California,
Los Alamos, New Mexico 87545

ABSTRACT

The modeling of manufacturing processes can be cast in a form which relies heavily on stores to and draws from object-oriented inventories, which contain the functionalities imposed on them by the other objects (including other inventories) in the model. These concepts have been implemented, but with some difficulties, for the particular case of pyrochemical operations at the DOE's Rocky Flats Plant using KEE, a frame-oriented expert system shell. An alternative implementation approach using CLOS (the now-standard Common Lisp Object System) has been briefly explored and was found to give significant simplifications. In preparation for a more extensive migration toward CLOS programming, we have implemented a useful subset of CLOS on top of the KEE shell.

I. INTRODUCTION AND BACKGROUND

A manufacturing process involves draws from a number of inventories of different types—inventories for materials and resources—and it eventually stores products and residues to appropriate inventories and returns resources to their inventories. The inventories may be concrete (e.g., a supply of chemical beakers) or abstract (e.g., a recording of operator exposure to hazardous materials).

Inventories can play an even greater role in process modeling when one allows them to carry their own functionality. For example, one task that might be performed by an inventory is keeping a history of its draws and stores. Or, a draw request on some inventory might trigger other actions, such as calling for a draw from another, related inventory or for starting up a whole new, related production process.

Placing functionality in inventories is an object-oriented programming (OOP) approach (e.g., COX 1987) to the simulation of the manufacturing processes. The general OOP description of a manufacturing plant might also involve objects representing a foreman (for decision-making), a controller queue, workcenters and parts, as well as inventories. In a working simulation there would be generic class-objects which would be fleshed out with member-instances, such as particular inventories or workcenters. The instance-objects communicate with one another by passing messages; an object receiving a message chooses to deal with that request according to coded methods incorporated in the data structure for the object itself.

At LANL we have undertaken a discrete-event simulation of the pyrochemical manufacturing processes at the DOE's Rocky Flats Complex (HODGE, SILBAR and KNUDSEN 1990a). In this work we worked in the OOP paradigm (HODGE, SILBAR and KNUDSEN 1990a), testing the concept of object-oriented inventories discussed above (SILBAR *et al.* 1990). Our initial prototype was implemented using Sun-4 workstations running the KEE expert system shell (INTELLICORP 1989).

In the following, Section II gives a brief discussion the general types of inventories needed for process simulation. Section III goes into the KEE implementation of the generic inventory classes in more detail, laying out functionalities, slots, and inherited behaviors and some of the implementation issues we addressed. Section IV describes how many of the problems found in our KEE implementation can be avoided using CLOS. The last major section discusses how we implemented a substantial subset of the CLOS standard within the KEE environment. This provides a programmer the option of developing his or her model simulations in a higher level, more disciplined language. The paper closes with a summary and notes some questions to be addressed in future work.

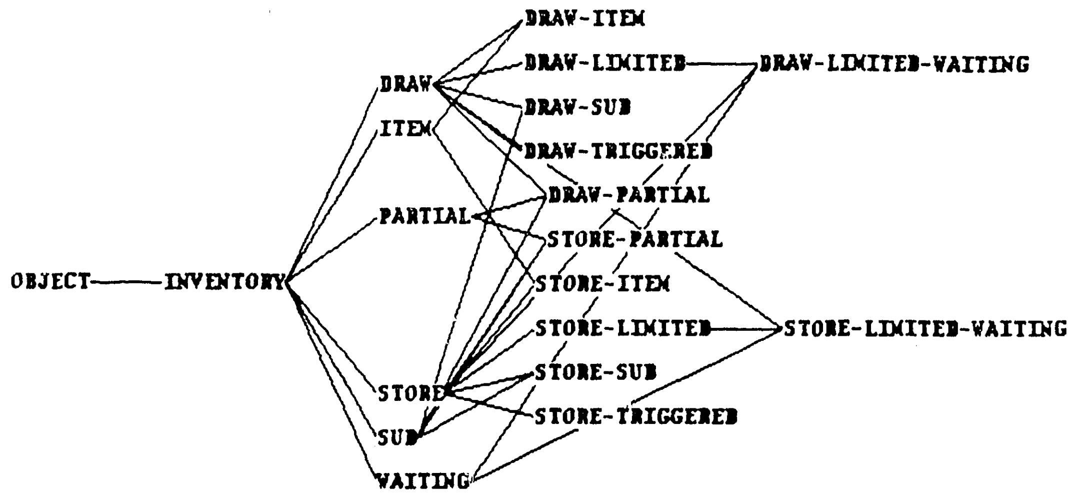

Fig. 1. Hierarchy of inventory classes. (Tangle graph created using KEE.)

II. GENERIC INVENTORY CLASSES

In brief, inventories should inherit their behavior from the following set of inventory classes. More details, along with examples, are given elsewhere (SILBAR *et al.* 1990).

Simple Draws and Stores – These inventories simply contain some bulk amount of a material or resource, and a draw or a store just decrements or increments the inventory level (a number). These simple inventories have no limits on the quantities drawn or stored. We need to distinguish a draw function from a "negative store" because a given inventory often needs to differentiate between these two functions and because they can involve different arguments and side-effects (see below).

Sub-Inventories – Inventories for which, say, a store must also increment some parent inventory. In fact, there might be a whole hierarchy of sub-inventories contained by higher-level inventories.

Item inventories – Inventories which track individual parts (which might be complicated structures in their own right) rather than a bulk amount.

Limited inventories – Inventories which have underflow or overflow functions which are invoked when a draw or store request bumps into a floor or ceiling. One cannot store more than there is capacity to store, nor can one draw more items than there are.

Waiting-List inventories – For certain critical resources—such as a particular kind of equipment, material, or storage space—a process may have to wait until that resource becomes available. Such inventories maintain waiting lists for those processes which have made unsatisfied requests. When a subsequent store or draw makes the resource available, the (oldest waiting) process is informed to make its request again.

Partial inventories – Inventories that accumulate a bulk amount that will eventually form a complete unit (e.g., residues which are packed in a drum). Such inventories typically pass the completed unit along to a parent item-inventory and re-initialize themselves to start a new unit.

Trigger inventories – Inventories which invoke some special action when a threshold is reached. There may well be several such thresholds and response functions for such an inventory.

On top of all these inventories is a generic top-level object, of which all inventories are subclasses. Figure 1 shows the class hierarchy for these general classes of inventories and how they inherit functionality from one another. Note the doubling of types for draws and stores.

Functionalities are not only inherited by, but can be *compounded* by subclass inventories. As a result, behavior tends to become more complex the lower down the hierarchical tree one goes. Figure 1 shows the multiple parentage of the generic inventory classes; Store-Partial, e.g., is a subclass of the Store, Partial and Sub inventory classes. Inheritance of behavior from multiple parents allows us to exploit the existing technology of flavor-mixing and/or wrappers.

Not shown in this hierarchical diagram are any inventory instances. In the RFP pyrochemistry model, there are about 75 different inventory instances. Many (if not most) of these inventory instances are a *mix* of some number of the generic inventory classes shown in Fig. 1. For example, the inventory named MSE-FURNACES is an example of a Draw-Limited-Waiting inventory (of an equipment resource) which inherits behavior from the Draw-Item, Draw-Limited, Draw, Item, and Waiting classes. It is also a Store inventory; otherwise there is no sense waiting for a furnace to become available. It happens in fact to be a Store-Item inventory.

III. THE KEE IMPLEMENTATION

The functionality of an inventory, in our model of the RFP manufacturing processes, is largely assembled through inheritance of behavior filtering down through the hierarchy of class objects to the member instances. That is, a given inventory is usually completely specified by assigning it as a member instance of some set of parent inventory classes (although, in principle, a given functionality for an inventory instance could have its primary method overwritten with its own special function). The following describes some details of how this was done in the framework of the KEE software.

First, OBJECT provides two accessor methods, GET-ATTRIBUTE and SET-ATTRIBUTE, for reading and writing slot values. These methods are also available to any child of OBJECT. (In practice, only those attributes that have been declared "public" can be accessed this way. This allows the programmer to reserve some "private" slots for internal use.) Further down the hierarchical inventory tree there are methods for other functionalities, such as GET-AVAILABLE-INVENTORY, DRAW, etc.

In KEE, methods are stored in special "method slots", either as named LISP procedures or as explicit lambda forms. We chose to store all our methods in LISP files, which we compile, rather than in the KEE knowledge base itself. This allows us to have use of documentation strings, comments, and ease of maintenance and transportability. There is a draw-back to this, however; we were unable to take advantage of the KEE "wrapper-body macros". Because we were working with compiled defuns, it was necessary to restrict our coding to "before" and "after" wrappers. These consisted of a few lines of code (containing compiled defuns) that were then inserted with the proper KEE syntax in the respective method slot. The lack of KEE wrapper-bodies led to some complexity in the logic of storing to and drawing from inventories.

Store and Draw inventories were treated in a parallel fashion, except that the store method may require, as an argument, a list of items to be stored and that the draw method may return, in addition to a keyword :SUCCESS and the quantity drawn, a list of the items drawn. To simplify the following discussion, we discuss only the case of drawing. Storing to an inventory is handled in a similar way.

Consider the case of a draw-inventory instance which is a member of several different inventory classes, i.e., an inventory which has a "wrapped" draw function. There are two major methods involved in drawing from such an inventory, a predicate called DRAW-FAILS? and the DRAW function itself. As the names imply, the first method checks to see if a draw is possible and the other actually performs the draw.

The DRAW-FAILS? method consists of a basic function that is performed by every invocation of the method plus some "before-wrappers" for handling the mix of constraints that must be checked before a draw can occur. (DRAW-FAILS? is, in fact, always called every time DRAW is called.) To simplify program logic (within the constraints of the KEE software), DRAW-FAILS? has, by fiat, no after-wrappers. The method returns nil if it is all right to draw, i.e., all the constraints on this inventory can be met. Otherwise, DRAW-FAILS? returns a list of keywords which indicate where the draw would fail and why. For example, the return value might indicate a failure to draw because the inventory's parent inventory is of the Draw-Limited type and the draw would drop that parent's inventory level below a floor. These keywords can be very useful for development and debugging purposes, as well as for the planning that other objects in the simulation model might undertake in the case of a failure.

The DRAW-FAILS? method has an optional argument SIDE-EFFECTS, which, if nil (the default value), means that DRAW-FAILS? acts as a pure, standalone predicate. If SIDE-EFFECTS is set to t, however, the method accumulates a list of side-effect actions that will be performed by the generic DRAW method if and only if *all* the DRAW-FAILS? before-wrappers return nil (i.e., there are no failures). That list is stored in a private slot (in each inventory involved), A-TO-EVALUATE-IF-OK, so those side-effect actions will be available to the subsequent DRAW message.

As an example, a Draw-Sub inventory will put a message on A-TO-EVALUATE-IF-OK to carry out the draw from its parent inventory. Similarly, a Draw-Item inventory puts on A-TO-EVALUATE-IF-OK a function which removes an item from the inventory item-list, checking that the number of items in that list is consistent with the inventory level (the number of items).

On the other hand, the DRAW method is often just the generic version and contains only after-wrappers, if any. There are in fact only two cases:

For Trigger inventories, the after-wrapper checks to see if a threshold has been reached or passed. If so, it then carries out the particular response function (defined separately in the methods file) associated with that threshold.

For a Draw-Limited-Waiting inventory, a successful store may allow some waiting process to have its draw request serviced. If so, that waiting item is removed from the list and a "run" message is sent to the waiting process. The sleeping process awakes and attempts another draw (which should now be successful).

After decrementing the inventory level, the main DRAW method evaluates each side-effect function put in the A-TO-EVALUATE-IF-OK list by DRAW-FAILS?. On exit, DRAW also resets A-TO-EVALUATE-IF-OK to nil in preparation for the next draw request.

For calls to DRAW from parents of sub-inventories, which must be handled with some care, an optional boolean argument FAILURE-CHECK (which is t by default) can be set to nil to avoid re-invoking the DRAW-FAILS? method with its SIDE-EFFECTS argument set to t. This avoids over-drawing grandparent inventories.

Most of the above complication involving private slots and boolean arguments results from the inability to use KEE WRAPPERBODYs programmatically, that is to say, with compiled defuns defined in a methods file. This was a disappointment to us, since the ability to do so would have been very useful for checking, e.g., whether the conditions to be satisfied for a successful draw held, and if so, completing that draw. However, a WRAPPERBODY in KEE is not a true function but a special marker which is replaced by the KEE method combination mechanism. One therefore cannot simply replace it with a defun name and have the arguments for the composed method come out properly. (WRAPPERBODY gets evaluated twice.) This is not a problem for BEFORE and AFTER wrappers in KEE, just for WRAPPERBODYs. In fact WRAPPERBODYs work well when the coding is entered directly into the method slots of a KEE knowledge base as lambda forms. Having to "handcraft" wrapped methods, however, does not fit well into our design decision to use compiled methods files and to build and load the KEE knowledge bases programmatically. This is, to a large extent, why we decided to use two methods, DRAW-FAILS? and DRAW, as described above.

Another complication of the KEE software forced us to keep the inheritance tree for methods relatively shallow. This was for the following two reasons. The DRAW-FAILS? before-wrapper for Draw-Limited-Waiting, for example, will be performed before that of its parent, Draw-Limited. This may not be what the programmer/developer always wants. Also, having most nesting go to only two levels, as in Fig. 1, gives the programmer better control over what is being done and when. (At an earlier stage of our development, we had considered Draw-Partial to be a subclass of Draw-Sub.)

IV. A CLOS IMPLEMENTATION

As we have seen in the last section, the problem with the present KEE implementation is that the inability to use KEE WRAPPERBODYs programatically forces us to write an an extra method, CAN-DRAW?. This function checks the constraints that a particular inventory instance has to satisfy, such as whether it can draw from a parent inventory or hits a floor or ceiling. CAN-DRAW? writes out, to private slots, error messages if it can not draw and, if it can, the side-effects that are to be evaluated.

It appears there can be considerable simplifications in the coding of the inventory class hierarchy using CLOS (e.g., BOBROW *et al.* 1988, STEELE 1990) over the present version written using the frame architecture of the KEE shell. As an experiment, we tried to see how things would look in a CLOS implementation of inventory classes. The test code included definition of the Inventory, Limited-Inventory, and Sub-Inventory classes and the draws and stores to/from them. (We did not bother trying to include functionality for recovering histories and the like; there should be no problems in doing so, if desired.)

The basic point is that, because of the ability in CLOS to invoke call-next-method, things become much cleaner and easier to read. There is no need to invoke a DRAW-FAILS? sub-call at all (although one might wish one in any case). Nor is there any need for the private slots A-FAILURE-LIST and A-TO-EVALUATE-IF-OK. These simplifications are illustrated by the following code fragments for the DRAW generic function:

```
(defgeneric draw (inv amt))

(defmethod draw
          ((inv inventory) amt)
  (decf (level inv) amt)
  '(:success ,(name inv) draw ,amt))

(defmethod draw
          ((inv limited-inventory) amt)
  (if (< (- (level inv) amt) (inv-floor inv))
      '(:failure :draw-hit-floor ,(name inv))
    (call-next-method)))

(defmethod draw
          ((inv sub-inventory) amt)
  (let* ((draw-parent (draw (parent inv) amt))
         (retpar (car draw-parent))
         (restpar (cdr draw-parent)))
    (if (eql retpar :failure)
        '(:failure :cannot-draw-parent
                   ,(name inv) ,restpar)
      (call-next-method))))
```

where the functions level, inv-floor, and parent are CLOS accessors for those slot-values (defined in the appropriate defclass statements).

The simplicity of the above code, compared with the KEE version we implemented first and discussed at length above, suggests that generic inventory classes implemented in CLOS would be both simpler to explain and to maintain.

V. IMPLEMENTING CLOS ON KEE

Motivated to some extent by the desire to use the newer CLOS syntax, we came to consider how one might integrate it with KEE. Eventually we realized that, because they had similar approaches to method inheritance and to method combination, the two different-appearing programming styles could indeed be largely reconciled. This line of thinking then evolved into an *implementation* of the CLOS language and syntax on top of the KEE shell (EGDORF 1990). This has the obvious advantage of retaining all the other useful features of KEE, such as the graphics and rule-reasoning capabilities. It is also optional; the programmer need only use the CLOS super-structure if he desires to.

We now describe briefly how the CLOS syntax is mapped onto the KEE core-functions, indicating some of the limitations of our KEE implementation.

First, a subset of the CLOS meta-object protocol is defined. Every class (defined by the CLOS function defclass) is an instance (i.e., a member-child) of Standard-Class or one of its subclasses. Moreover, every such class is also a subclass of Standard-Object.

The CLOS construct (defclass ...) is built on top of KEE'S core-function (create.unit ...). CLOS slots are slots in the KEE unit representing a class which will have instances (member units) defined later. The :accessor functions are limited, being built as defuns rather than true generic functions. also, one class option, :default-initargs, and some slot options, ":allocation :class" and ":initargs", are not supported in the present implementation.

The CLOS (defgeneric ...) construction simply turns into KEE's (unitmsg ...). KEE does all the work of the method combination. A generic function is not automatically created, however, by this version of (defmethod ...); the programmer must explicitly define the (defgeneric ...) beforehand. The CLOS options :documentation and :method are supported.

The (defmethod ...) is defined as a macro which adds LISP forms to the corresponding method slot in a KEE unit. KEE performs the task of method combination in its own way. In contrast to full CLOS, only the first parameter is specialized. This reflects KEE's ownership of methods by a class. Because the implementation does not try to compile the (combined) methods, (call-next-method) is simple, being implemented as a KEE WRAPPERBODY.

Two other CLOS functions that are indispensible are `make-instance` and `slot-value`. These are defined using KEE's `create.unit` and `get.value` (or `put.value` in the case of a setf function), respectively.

We have recently reformulated our RFP model using this CLOS implementation on top of KEE. (In the process we have de-emphasized the *central* role of inventories and given more emphasis to an event-architecture style for the discrete-event simulation.) Our experience has been quite positive. The lack of `:initargs` and other parts of CLOS not in our implementation is not crippling. As a benefit, the code size of the model is much smaller than that using KEE core-functions and, perhaps, the learning time for a newcomer to the programming style is shorter. The implementation certainly "works" in the small test cases we have built to now. And, at the least, use of this enhancement now should ease any future migration from KEE, a proprietary product nearing the end of its supported life, to a more standard CLOS programming environment.

VI. Summary

The main conclusion of this paper is that an implementation of our object-oriented inventory classes would have been much easier in CLOS than in KEE. However, there are many other reasons, e.g., the graphics capabilities, why we use KEE for our model simulations besides object-oriented programming. We are not yet ready to abandon our use of this richly featured expert system shell. In fact, as we have shown in the last section, it is possible to extend KEE so it *implements* a significant subset CLOS. This, we feel, is a very useful enhancement of KEE that may be of interest to the community at large.

REFERENCES

D. G. Bobrow *et al.*, "Common Lisp Object System Specification", X3J13 Document 88-002R (June 1988).

B. J. Cox, *Object-Oriented Programming—An Evolutionary Approach*, Addison-Wesley (1987).

H. W. Egdorf, "A CLOS Implementation on Top of KEE", presented at Conf. on AI in the DOE Complex, Idaho Falls ID, October 1990.

C. A. Hodge, R. R. Silbar, and P. D. Knudsen, (a) "Modeling Nuclear Materials Processes", annual meeting of the Inst. for Nuclear Materials Management, Los Angeles CA, July 1990. (b) "Interaction of Objects in Manufacturing Process Simulation", Simulation Workshop, bi-annual meeting of the Am. Assoc. of Artificial Intelligence, Boston MA, July 1990.

IntelliCorp, Inc., *Knowledge Engineering Environment*, a software package referred to as KEE, June 1989. KEE is a trademark of IntelliCorp, Inc. (Mountain View CA) and is based on Sun Common Lisp, provided for Sun workstations by Lucid, Inc. (Menlo Park CA).

R. R. Silbar, P. D. Knudsen, C. A. Hodge, and J. W. Jackson, "Object-Oriented Inventories for Simulations of Manufacturing Processes", Proc. of the Conf. on Artifical Intelligence Systems in Government, Washington DC, May 1990.

G. L. Steele, *Common Lisp: the Language*, 2nd Ed., Digital Press (1990).

END

DATE FILMED

12 / 04 / 90