

# Four Models of Job Control

David Flater  
Edward Barkmeyer  
Evan Wallace

Bldg. 220 Rm. A127  
National Institute of Standards and Technology  
Gaithersburg, MD 20899 U.S.A.

September 19, 1997

## **Abstract**

In a distributed, object-oriented, hierarchical shop control system, there are many ways of organizing the interactions between supervisory controllers and subordinate controllers. Depending on what model of job control is used, objects will be allocated differently among the levels of control, and the interactions between controllers will vary considerably. This paper describes four models of job control that are equally functional, but use different paradigms for managing jobs. Although each is attempting to serve the same purpose in the same context, the four ways of viewing the system lead to distinct implementations which cannot interoperate. This demonstrates the importance of specifying the control model when designing an object-oriented manufacturing system.

## **1 Preface**

Some years ago, object-oriented programming languages were introduced to improve the maintainability and reusability of software systems. The object-oriented approach permits developers to map real-world entities, such as machine controllers, to programming language constructs called *classes* that encapsulate the attributes and behaviors needed to model the real-world entity. This usually results in software that is more understandable as well as more maintainable and reusable.

Object-oriented programming languages are complemented by an object-oriented mechanism for communication between programs, where the programs run independently and may run on different

physical computer systems. The “owner” program for a given object creates the object and uses the program memory resources to store the state (data) of the object. When a program invokes an operation on an object that it owns (a “local” object), it is just like a function call; but when a program invokes an operation on an object that is owned by a different program (a “remote” object), the request and its response must be transmitted over the network. Distributed object architectures, such as the Common Object Request Broker Architecture (CORBA) (OMG 1997), provide this network service so that invoking operations on remote objects is no harder for the programmer than invoking operations on local objects. This capability leads to the construction of “distributed object systems” in which the conceptual design of a distributed system is mapped onto a distributed object implementation. Because of their maintainability, performance, and standardization, distributed object architectures are being examined by some manufacturers as a possible replacement for aging and/or inadequate message bus architectures.

Object orientation is not, however, without its problems. A good mapping between real-world entities and implementation classes only happens as the result of good design decisions. If the design embodies confused thinking, then the resulting implementation will suffer, and may not work out at all. A consistent way of viewing a system is called an “object model.” There may be several equally good object models for a particular system, but a consistent design requires that exactly one object model be chosen. Sometimes it is not until implementation is underway, and unexpected incompatibilities between components begin to crop up, that it is found that the system design does not consistently adhere to one object model.

This paper describes four different object models for hierarchical job control in manufacturing. In hierarchical job control, planning and dispatching responsibility is distributed between supervisory entities and subordinate entities. A job supervisor dispatches low-level jobs to any number of subordinates to accomplish its high-level goals. This permits localization of planning and replanning activities and easier coordination of the manufacturing activities of interrelated equipment.

Hierarchical job control is a good candidate for distributed object technology because it is decentralized and uses multiple semi-autonomous entities. By presenting four object models for hierarchical job control, we hope to facilitate the design process for distributed object systems in the manufacturing domain.

## 2 Introduction

For the National Advanced Manufacturing Testbed (NAMT) Framework project (Bloom and Christopher 1996), a prototype, distributed manufacturing system was developed at the National Institute

of Standards and Technology. It serves as a testbed and trial implementation for emerging industry-developed specifications such as SEMATECH's CIM Framework (CIMF) (Eng *et al.* 1996).

In the testbed system, one of the important areas for protocol testing is the management of Jobs (or Tasks) on the factory floor. Following the apparent intent of the CIMF and several other specifications now being developed for "Manufacturing Execution Systems," the overall model used for the management of Jobs is a hierarchical one. There are two components involved in the communications about a given Job – a "supervisor," such as a shop dispatcher, and a "subordinate," such as a workcell controller.

During the initial analysis in which the Framework team attempted to understand the component interactions of version 1.3 of the CIMF, three possible models of job control – three different but equally valid scenarios for component interaction – came to be identified. Their names correspond to the color of ink that was used to characterize them on the whiteboard. The fourth model, the "database" model, was identified later.

Each model admits to several possible implementation structures, because functions can be assigned to different classes of objects (where classes of objects are internal software structures) within the same agent to accomplish the same effect. For this discussion, an *agent* is one of the entities that communicate with each other in a distributed manufacturing system. A single agent may implement multiple classes of objects, but ultimately those objects will all share the same process space on the same machine. Therefore, at a certain level, it does not matter which specific object handles any given operation, since they are all different faces of the same agent. Depending on whether the goal is maximal similarity to a specification, adherence to some particular methodology, or ease of implementation, the allocation of operations to specific objects will be made in different ways. That which distinguishes one model of job control from another is the allocation of objects to different agents, not the allocation of operations to different objects within an agent.

The following sections present four generic models of job control and identify possible implementation variants. Each one is illustrated with an example showing two levels of control, implemented using two generic classes, Controller and Job.

### 3 The Green Model

The Green model corresponds to a way of thinking in which a job is "owned" by the subordinate. (This "ownership" means that, in an implementation, the Job object will exist in the subordinate's work space and may be tightly integrated with the subordinate's control system.) The subordinate agent provides one Controller object and a Job object to represent each task; the supervisor provides

at most a Controller object (to the subordinate). Task-related feedback to the supervisor must either be sent directly to the Controller object or communicated through an out-of-band mechanism, such as “events.”

A normal job dispatch would occur as follows:

1. Supervisor invokes a `createJob` operation on the subordinate Controller, which returns a Job object owned by the subordinate.
2. Supervisor supplies values for the attributes of the Job object to describe the task that must be done.
3. Supervisor instructs the subordinate to start the job. Implementation variants:
  - (a) Operation on the Job, `Job::start(void)`.
  - (b) Operation on the subordinate Controller, `Controller::startJob(Job)`.
4. Subordinate begins the task.
5. If the supervisor needs to abort the task, or perform some other supervisory intervention (pause/resume, stop, hold, etc.), it does so in a way similar to the way it started the job; e.g.:
  - (a) Operation on the Job, `Job::abort(void)`, or
  - (b) Operation on the subordinate Controller, `Controller::abortJob(Job)`.
6. When the subordinate needs to report a change of state in the assigned Job, there are three principal approaches:
  - (a) Subordinate invokes an “inform” Operation on the supervisory Controller, `Controller::informJobCompleted(jobName)`. (Only the job *name* is meaningful to the supervisor, since the job itself is owned by the subordinate.)
  - (b) Supervisor polls the subordinate for information, e.g. `Job::get_status(void)`, or `Controller::get_status(Job)`.
  - (c) Subordinate notifies the supervisor through an out-of-band mechanism, such as “events.”

The latter two mechanisms for feedback to the supervisor do not require the supervisor to expose *any* object services to the subordinate. The Green model thus offers the potential for the supervisor to be a “pure client” – an agent that exposes no services – and the subordinate to be a “pure server” – an agent that does not use the services of any other agent. This makes the Green model easiest to

implement from a programmer's point of view. It also makes Green the closest of the four models to an open-loop control system in which the subordinate provides no feedback at all.

The choice between using commands of the form `Job::abort(void)` or `Controller::abortJob(Job)` corresponds to two different approaches to objectifying the job control domain. If one recognizes jobs as tangible objects having state, behavior, and identity (Booch 1991), then pausing, stopping, and aborting are all *behaviors* of jobs, and they belong as operations on the Job objects. On the other hand, one could argue that jobs are not tangible objects, but are merely processes performed by the controllers. In that case, the pausing, stopping, and aborting of jobs are all behaviors of the Controller object, and the Jobs themselves, if they exist at all, are only records for job-related data. The latter approach is actually easier to implement, since it requires less inter-object communication.

Of the three feedback mechanisms suggested, only the second is really consistent with object-oriented design, in that an operation dealing with the status of the Job is an operation on the Job object or the corresponding Controller object. But the polling mechanism thus represented is very inefficient for real-time control activities. The “inform” mechanism, using the Job name, is a kind of *faux object-oriented* mechanism that is adequate to the need and maintains at least the client/server relationship. But the “notification” mechanism poses a real threat to object-oriented design. The mechanism itself is simply an open-ended messaging scheme that maintains neither the object-oriented association of the operation with the Job object nor the client/server relationship. It is a general-purpose communication service that is “object-oriented” only to the extent that the supervisor internally renders the arrival of the event into one of the other two forms.

## 4 The Black Model

The Black model corresponds to a way of thinking in which a job is “owned” by the supervisor and is “assigned” to the subordinate to work on. The supervisory agent provides one Controller object and a Job object to represent each subordinate task; the subordinate provides only a Controller object. Task-related feedback to the supervisor may be communicated through the Job object that was assigned to the subordinate.

A normal job dispatch would occur as follows:

1. Supervisor creates a Job and fills it with information.
2. Supervisor assigns the Job to the subordinate, invoking `startJob(Job)` on the subordinate Controller object.
3. Subordinate begins the task.

4. If the supervisor needs to abort the task, or perform some other supervisory intervention (pause/resume, stop, hold, etc.), it does so by invoking `<operation>Job(Job)`, e.g. `abortJob(Job)`, on the subordinate Controller object.
5. When the subordinate needs to report changes in the job state to the supervisor, it uses one of the following:
  - (a) Operation on the Job, `Job::setState(Completed)`.
  - (b) Operation on the supervisory Controller, `Controller::informJobCompleted(Job)`. (Note that in this model, the Job object itself can be referenced, where in the Green model only the job *name* was meaningful.)
  - (c) Out-of-band Events, as in the Green model.

The advantage of the Black model is that it exposes the Job object in the supervisor, giving the subordinate a truly object-oriented means of reporting status changes and providing other feedback information to the supervisor. Because these operations invoke the supervisor to perform some action, this model effectively addresses the need for real-time interactions.

The disadvantage is that the Black model makes both supervisor and subordinate simultaneously both clients and servers. This can significantly complicate the actual programming task for these agents.

## 5 The Red Model

The Red model corresponds to a way of thinking in which a job is not “owned” by either the supervisor or the subordinate, but rather is a joint concept of which each has a somewhat independent view. Each agent has a Job object that represents *its* view of the task, and each identifies its Job object to the other. (This is closer to the internal models of the two agents, since the supervisor’s Job object is related to other tasks and concerns in the supervisor’s scope of work, while the subordinate’s Job object is related to the details of its execution within the subordinate’s scope of work.) The agents communicate with each other and “maintain synchronization” of the two Job objects primarily by invoking operations on each other’s Job object.

A normal job dispatch would occur as follows:

1. Supervisor creates a JobForS and fills it with information.
2. Supervisor assigns the JobForS to the subordinate, `startJob(JobForS)`. Subordinate creates its own view of the task (a JobAtS) and returns the JobAtS identifier to the supervisor.

3. Subordinate begins the task.
4. If the supervisor needs to abort the task, or perform some other supervisory intervention (pause/resume, stop, hold, etc.), it does so in one of two ways:
  - (a) Operation on the subordinate Job, e.g. `JobAtS::abort(void)`.
  - (b) Operation on the subordinate Controller, e.g. `Controller::abortJob(JobAtS)`.
5. When the subordinate needs to report a change of state in the assigned Job, providing feedback to the supervisor, it does so by one of:
  - (a) Operation on the supervisor Job, `JobForS::setState(Completed)`.
  - (b) Operation on the supervisory Controller, `Controller::informJobCompleted(JobForS)`. (As in the Black model, the subordinate can provide a reference to the supervisor Job object.)
  - (c) Out-of-band Events, as in the Green model.

The advantage of the Red model is that each agent exposes its view of the job and each agent has access to operations on the other agent's Job. These can be used for retrieving information as well as sending commands or notifications. Operations on Jobs can be used consistently with no need for job control operations on the Controller or out-of-band feedback mechanisms. And because each can invoke the other, real-time interaction requirements can be met.

The disadvantage is the same as that of the Black model: the Red model makes both supervisor and subordinate simultaneously both clients and servers, and this can significantly complicate the actual programming task for these agents.

Apart from object-oriented purism, there is a further advantage to the Red model over the Black model when one expands the view to include external monitoring of shop-floor activity. With the Black model, the external monitor client can only obtain the *supervisor's* view of what is happening to the Job; it cannot obtain first-hand Job information from the subordinate that is actually carrying out the job. With the Red model, the external monitor can obtain both, but most importantly, it can obtain the *subordinate's* view.

The Red model is thus the synthesis of the best features of both the Black model and the Green model: convenient object-oriented real-time interaction coupled with external visibility of the subordinate's Job. Unfortunately, it is also the most complicated in terms of interface and programming requirements.

## 6 The Database Model

The perspective that the job is really not inherent in either the supervisor or the subordinate but is a collective abstraction suggests that Job objects could be “owned” by a “third-party” database-like server that is simply tracking the state and characteristics of the job.

In this case, the supervisor and the subordinate would each provide only a Controller object, and the “database server” would provide a JobManager object and one Job object for each job anywhere in the system.

With this model, a normal job dispatch would occur as follows:

1. Supervisor creates a Job and fills it with information: `JobManager::createJob(...)`, etc.
2. Supervisor assigns the Job to the subordinate, invoking: `Controller::startJob(Job)` on the subordinate Controller object.
3. Subordinate begins the task.
4. If the supervisor needs to abort the task, or perform some other supervisory intervention (pause/resume, stop, hold, etc.), it does so by invoking `<operation>Job(Job)`, e.g. `abortJob(Job)`, on the subordinate Controller object.
5. When the subordinate needs to report changes in the job state to the supervisor, it uses *both*:
  - (a) Operation on the Job, `Job::setState(Completed)`, and
  - (b) Operation on the supervisory Controller, `Controller::informJobStateChange(Job)`.

This model has all the advantages of the Red model with somewhat less complexity and slightly higher overhead. In addition, it separates the issue of who “owns” the Job objects from how the interaction is specified. It supports reassignment of the JobManager functions from one implementation to another. In fact, the agent that actually has the Job object need not be a third party. It could be the supervisor, or even the subordinate, although the last is undesirable because the JobManager should have *all* the Job objects in the system.

This model also supports a data-driven approach that can be consistently applied throughout the system, where all state information is kept in a single database and all decisions are made based on the contents of that database. That approach makes it easier to keep a history and to implement persistence for important information. Taking this view further, the integrity constraints and triggers provided by modern “active” databases could be put to good use insuring that the supervisor’s and the subordinate’s views of the job are consistent, deriving the states of higher-level jobs from the

states of their constituent tasks, and automatically notifying the controllers of important changes. At some point, however, this shifts significant control functionality away from the systems that were designed to make such decisions and wrecks the hierarchical control architecture, potentially creating a monolithic controller for the entire shop.

The separability advantage can also be a disadvantage: the model does not make clear to the programmer of any agent whether that agent is expected to provide the Job objects and services. This means that the client behavior of agents, as well as their server behavior must be defined. For example, the shop dispatcher must be *required* to use the external JobManager services for createJob() even when the agent actually contains an implementation of those services.

But the most serious disadvantage is a practical one: when the JobManager is a third party, there is a third system, together with all its supports and communications, that can fail and prevent the two “functional” Controllers from getting work done. Overall, this represents an additional single point of failure for the entire factory.

## 7 Conclusion

Four different but equally functional models of job control have been presented. Although each is attempting to serve the same purpose in the same context, the four ways of viewing the system lead to distinct implementations which cannot interoperate. It is not possible to use a supervisor from one model with a subordinate from another; they would have conflicting assumptions about the ownership and usage of job objects.

Given these incompatibilities caused by nothing more than differing models of job control, it is essential that any standard or specification within the job control arena be clear and unambiguous in its description of its job control model to avoid potential interoperability failures. Definitions of controller objects and job objects are necessary, but in the absence of a control model they are not sufficient to guarantee interoperability.

Even though the number of entities in our models is small, we still have not exhausted all possible models of job control. For example, one reviewer proposed the “mobile job” model, where the Job object is actually transferred from supervisor to subordinate and back again. As new paradigms like mobile objects emerge, new ways of performing old tasks become apparent – and specifications which once were unambiguous can become subject to new interpretations.

## 8 Acknowledgements

The authors thank all of the reviewers, within and without NIST, whose comments have improved this paper.

## References

BLOOM, H. M. AND CHRISTOPHER, N., 1996, A framework for distributed and virtual discrete part manufacturing. In *Proceedings of the CALS EXPO '96*, Long Beach, CA.

BOOCH, G., 1991, *Object-Oriented Design with Applications* (Benjamin/Cummings), p. 77.

ENG, L., FREED, K., HOLLISTER, J., JOBE, C., MCGUIRE, P., MOSER, A., PARIKH, V., PRATT, M., WASKIEWICZ, F., AND YEAGER, F., 1996, *Computer Integrated Manufacturing (CIM) Application Framework Specification 1.3* (SEMATECH, 2706 Montopolis Drive, Austin, TX 78741 U.S.A.).

OMG, 1997, *Object Management Group home page* (<URL:<http://www.omg.org/>>).