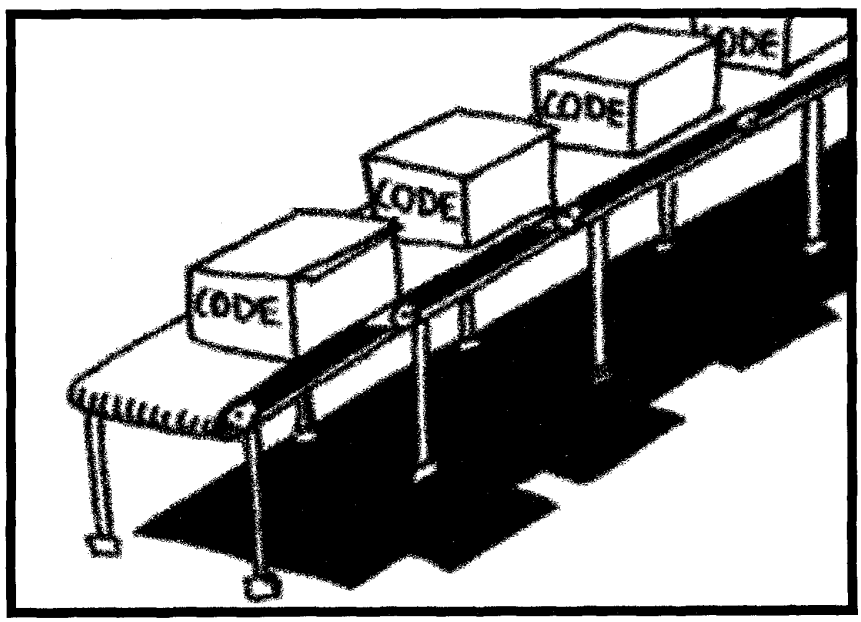
Creating Applications From Components: A Manufacturing Framework Design

The manufacturing-application framework described here lets you develop software from components without programming. The framework's design generalizes the class structure which models a fixed domain application through a sequence of transformations.



HANS ALBRECHT SCHMID
Fachhochschule Konstanz

Manufacturing systems for part-processing metal industries consist of components—such as machines, robots, and buffer stores—and a software control system. To build a manufacturing system, little design work is required to tailor the components to the intended configuration. The software control system, however, is usually developed from scratch, which is not only expensive and time-consuming but reuses little—if any—software. An off-the-shelf approach to software development, achieved by selecting and configuring reusable software components, would result in significant savings.

0740-7459/96/$05.00 © 1996 IEEE

Although creating applications from reusable components was first proposed in 1968,[1] it took more than a decade before reuse began to materialize. At the same time, object-oriented technology[2,3] emerged, providing a sound foundation for reuse. The reuse of domain-independent class libraries thus gained considerable momentum; reuse of domain-specific class libraries was pursued to a lesser degree. Both kinds of reuse, however, offer only a limited saving of development effort and time, largely because you must develop application logic for each application. In contrast, OO frameworks[4] let you reuse application logic.

DSBB frameworks offer significant savings of development cost and time.

I initiated and led the development, design, and structuring of a domain-specific black-box (DSBB) framework for automated part-processing manufacturing. We began development of the Open Software Framework for Manufacturing, called OSEFA, in 1993. An OSEFA prototype with strongly restricted flexibility has been running since early 1994, the full framework since late 1995, at the computer-integrated manufacturing model factory jointly operated by the computer science and mechanical engineering departments of the Fachhochschule of Konstanz, Germany.

DSBB frameworks contain components that model an application domain's entities, concepts, and logic so you can create an application from components. Thus, after the initial development investment, DSBB frameworks offer significant savings of development cost and time over from-scratch development.

FRAMEWORKS

A framework is a set of object classes that collaborate to carry out a set of responsibilities;[5] it reuses both the high-level design of a program and its implementation.[4] Frameworks are classified by application and domain and, depending on how an application is created, as either white-box or black-box.

Application framework. An application framework provides the basic functionality of a working application, which can be easily tailored to a specialized application. Most application frameworks, such as ET++[6] and Taligent's CommonPoint, provide basic administrative functionality, which is required "horizontally" over a range of application domains such as user interfaces and data administration. But because the specific content that models the application domain is missing, such frameworks might better be called *application-enabling frameworks.*

Domain-specific framework. Less common are *domain-specific frameworks,* which model the domain-specific functionality via enterprise objects and a generic application logic that often can be performed in many different domain configurations. With a domain-specific framework, you can create an application by implementing a specific configuration with enterprise objects and then binding the generic application logic to this configuration.

White and black boxes. *White-box frameworks,* such as the model view controller framework from Smalltalk-80, or ET++, are the original frameworks. They provide incomplete "frame" classes, usually as abstract classes, describing the allocation of responsibilities to components and their interfaces. You can specialize the framework by deriving application-specific classes from the frame classes through inheritance, and by completing or redefining their methods. One disadvantage is that you must have detailed framework code knowledge.

Black-box frameworks[7] let you create an application from components rather than programming. A black-box framework provides alternative (complete) classes for a variable aspect, known as a "hot spot,"[8] each class realizing one instance of the variability. For example, for the machining of parts, a black-box framework will provide different machine classes for different machines. You can thus select one or more suitable classes when creating an application. Thus, an application is developed by selecting, parameterizing, and configuring its components. This does not require knowledge of framework code—and is thus easier to use. However, the development of black-box frameworks requires more careful planning for reuse and more development effort.[9]

Hot-spot subsystem. Alternative classes of a black-box framework are organized in a hot-spot subsystem[9] to let you dynamically configure an application without recompilation. A hot-spot subsystem contains a base class defining the common responsibilities of the variable aspect and, in simple cases, only alternative subclasses derived from it (interface inheritance). In more complex cases, however, additional classes and relationships among the base class, the derived classes, and classes external to a hot-spot subsystem may exist. Usually, the classes and relationships are structured according to a design pattern.[10] This knowledge makes it easier for you to develop the structure of hot-spot subsystems.

When a class sends a message to an object of a hot-spot subsystem class, the message-requesting class refers to the service-providing object by a polymorphic reference typed with the base class. The polymorphic reference is set to one of the subclasses when configuring an application from the framework. Subclasses can be added to a hot-spot subsystem without requiring recompilation of the existing black-box framework.

MANUFACTURING APPLICATION DOMAIN

The lower portion of Figure 1 shows a simplified manufacturing cell[11] from the OSEFA domain, that produces axial-symmetric parts from raw parts in the cell store. The cell consists of a computerized numerical control (CNC) lathe, which machines raw parts, a pallet buffer store, a portal robot, and a store for pallets. The portal robot is used both to move pallets to and from a buffer store, using a pallet gripper, and to load and unload parts from a pallet on the buffer store into the lathe, using a part gripper.

Although the configuration of different cells in the manufacturing subdomain covered by OSEFA may differ—for example, a cell may contain several machines—the processing done by each machine is the same. An external system, such as those for material request planning or manual shop-floor planning, describes the mechanical processing to be performed in a cell by a list of machine orders for each machine and by a work sheet. A machine order describes the part (and quantity) to be machined. A worksheet entry details the resources required for a machining step. Each machine processes concurrently, in the same way, one machine order after another. A machine order is processed through a sequence of part-processing steps, which forms the central part of the application logic.

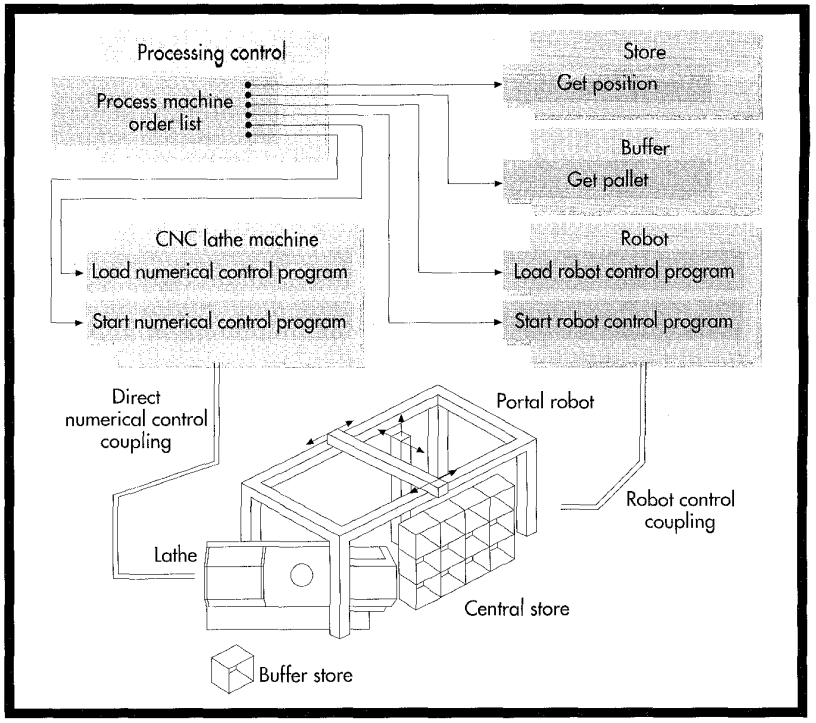


Figure 1. Entities of a manufacturing cell (lower part) and objects modeling them (upper part).

Abstract application logic. After the system reads the machine order and sets up required resources (such as numerical control programs), the system performs the abstract application logic in a seven-step sequence, simplified as follows.

1. A pallet with raw parts is transported from the store to the machine's buffer store.
2. A part is loaded from the raw-part pallet onto the machine.
3. The part is machined.
4. The part is unloaded from the machine onto the pallet.
5. Steps 2 to 4 are repeated until all parts on the pallet are processed.
6. The pallet is transported back to the central store.
7. Steps 1 to 6 are repeated until all pallets of a machine order are processed.

In reality, the application logic is more complex for two reasons: OSEFA is a reactive system, which means that each action is performed only when notified by an event that the preceding action was terminated. Further, the processing sequence is optimized so that, when possible, pallets can move concurrently with the machining step.

Configuration variability. All manufacturing cells and systems in the application subdomain are composed of a store and one or more machines, handling units, transport systems, and buffer stores. However, there are three primary areas of configuration variability that must be considered.

- *Variability of machines and devices.* These can consist of different but similar machines, and different kinds of machines (CNC machines, assembly robots, and programmable logic control machines or assembly stations); transport systems (portal robots, programmable logic controlled conveyor belts, or automatic guided vehicles); and handling systems.
- *Variability of machine and device associations.* For example, one or two handling devices may be attached to each machine for loading and unloading, or a single portal robot may load and unload all machines; the transport system and handling systems may be separate, or the portal robot may handle both functions.
- *Local variations of machining.* For example, different tools, such as a sin-

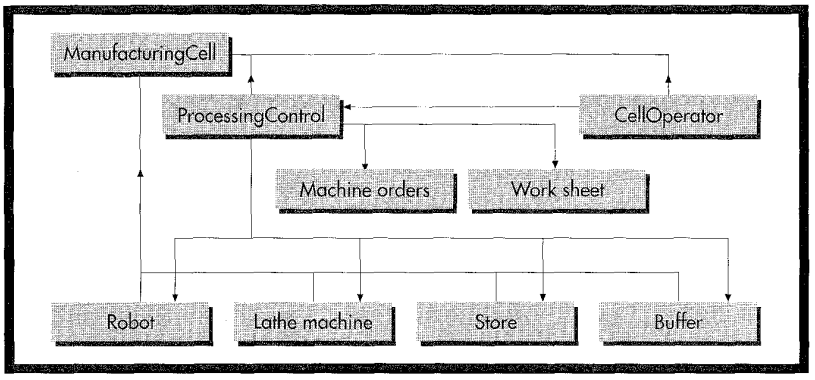


Figure 2. *Coad/Yourdon diagram of a manufacturing cell.*

gle or double gripper, can load a part into a machine; which tool is used depends on the machine order. Different local equipment and tools require different sequences for processing the parts, making it difficult to reuse the application logic.

The three areas of variability that affect the manufacturing subdomain are also characteristic of other domains. In many domains, there are

- the use of different, but similar devices;
- the use of different kinds of devices to achieve the same service for an application; and
- variations in a procedure or in the application logic.

Given this, if you know how to structure a framework for manufacturing that embodies these variable areas, you should be able to apply the same structuring techniques to similar application domains.

MANUFACTURING CELL: OO ANALYSIS

As the first step in our framework design, we developed the class structure of the control software for a specific manufacturing cell in the OSEFA subdomain, modeling the cell's entities using an OO analysis methodology.[2-3] The lathe machine, portal robot, and store were modeled by the objects shown in the upper portion of Figure 1. In addition, a ProcessingControl object modeled the workers' knowledge about parts processing.

An object is described and defined by its application-relevant dynamic behavior, that is, the actions it can perform in the application world. For example, a lathe machine object can load and execute numerical-control programs, among other services; a portal-robot object can load and execute robot control programs; and a store object can give the position of a pallet in the store. An object may request another object to provide a service specifying its details by additional information. For example, the ProcessingControl object may request a CNC lathe machine to load a numerical-control program with a given number.

Because a manufacturing control system is reactive, many objects (such as CNC machines or robots) are directly coupled to the real-world entities they model through communication lines, as Figure 1 shows. A coupled object can send a command to the entity it models to make it perform a service.

Classes and interrelationships. The objects and their interrelationships in Figure 1 can be represented more abstractly by the Coad/Yourdon diagram[3] in Figure 2. This shows the classes introduced in Figure 1, Lathe Machine, Robot, Store, and Buffer, as well as the object classes CellOperator, ProcessingControl (with the Machine Orders and Work Sheet information), and aggregate ManufacturingCell. The relationships among them are

- a ManufacturingCell contains (any number of) Store, Buffer, Lathe Machine, Robot, ProcessingControl, and CellOperator objects; and
- ProcessingControl requests services from these objects and provides services to the CellOperator object.

The derived class structure does not exhibit specific OO characteristics, such as specialization, generalization, polymorphism, or dynamic binding. Instead, it represents an object-based approach that builds on abstract data types and data encapsulation.

Concrete application logic. ProcessingControl performs a seven-step part-processing sequence that is a concrete instance of the abstract part-processing application logic described earlier.

1. To move the pallet with raw parts from the store to the buffer store, ProcessingControl sends a request to the robot to
   - load the robot program A;
   - load, as a parameter of program A, the pallet position in the store; and
   - start program A.
2. To load a part from the buffer store into the lathe machine, ProcessingControl sends a request to the robot to
   - load and start robot control program B, which exchanges the pallet gripper for a part gripper;
   - load the robot control program C, which loads the part;
   - load parameters of program C with the part position on the pallet; and
   - start program C.
3. After the robot has loaded the part, ProcessingControl sends a request to the lathe machine to start the numerical-control program that machines the part.
4. The part is unloaded onto the pallet, similar to step 2.
5. Steps 2 to 4 are repeated until all parts of the pallet are processed.
6. The processed pallet is transported back to the central store, similar to step 1.
7. Steps 1 to 6 are repeated until all pallets in the order are processed.

The concrete application logic is bound to the particular configuration of the sample cell. Requests for application-logic–related services—such as performing a transport task—are replaced by device-related service

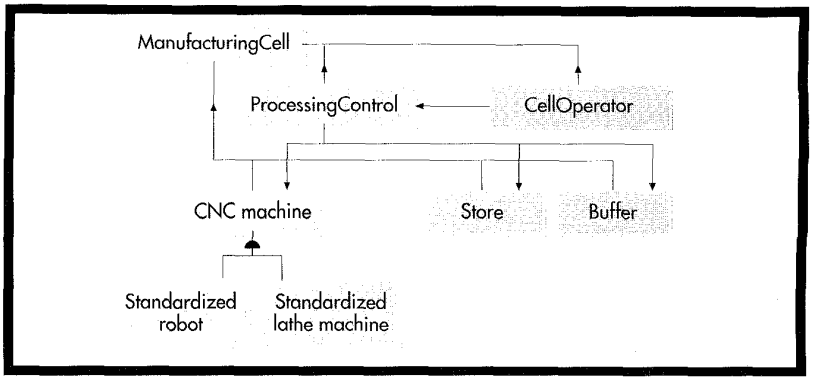


Figure 3. Generalizing machines with dynamic binding. The shaded area indicates a hot-spot subsystem.

requests, such as parameterizing (and starting) a robot control program.

Because these requests are bound to specific devices, ProcessingControl is clearly not reusable with different configurations. The result of an OO analysis without a domain analysis is a class structure with classes tailored to the specific cell configuration. Even for a simple configuration modification, such as exchanging machines that have different interfaces, ProcessingControl requires major modifications.

Consequently, reusability is still limited, and the derived class structure is unsuitable as the basis for a black-box framework. To make it reusable, we generalize the class structure by a sequence of transformations. Each transformation[9] introduces parts of the configuration variability identified earlier into the framework class structure.

FIRST TRANSFORMATION: DEVICE GENERALIZATION

This transformation deals with the simplest area of variability: the variability of similar machines and devices. To make ProcessingControl reusable when exchanging machines, all machines must have the same interface, and thus different machines must be generalized by common properties and services. The difficulty is in trying to establish a valid concept of a generalized machine. Its services must be defined such that they can be provided by—and implemented efficiently on—"every" machine (where "every" means "sufficiently many").

In OSEFA, we defined a generalized CNC-Machine class and, we believe, proved the validity of this concept. This class has properties common to different CNC machines and robots and includes all machines (from robots to coordinate-measurement machines) that are controlled by numerical control or robot programs. Services shared by these machines include

- uploading and downloading numerical-control and robot control programs, data, and parameters;
- starting and stopping a downloaded program; and
- interrogating the machine state;

Standardized interface. With these services we have a standardized interface that features the common functionality of CNC machines. This interface was realized by declaring an abstract base class CNC-Machine. From this base class, we derived subclasses modeling the real CNC machines. The CNC-Machine base class and subclasses form a hot-spot subsystem of a domain-specific black-box framework. According to the rules for structuring hot-spot subsystems, ManufacturingCell should contain polymorphic references, typed with the CNC-Machine base class, via which a message is sent. Figure 3 represents this as a "has-as-part" relationship from manufacturing cell to CNC machine. In this way, no recompilation is required when machines are exchanged. This is a distinct advantage of object orientation over object-based languages.

The references, which are set during the configuration process, refer to machines contained in the actual configuration. The configuration process must ensure that a reference meant to refer to a transport and handling device does not refer to a processing machine, and vice versa.

When doing a machining step, it makes no difference whether it is done on a milling machine or on a lathe machine, or that a processing step on a lathe was replaced by an assembly step with a robot. In all cases, the processing is exactly the same, both on an abstract level and on a service request level. Thus, we made ProcessingControl reusable relative to different CNC machines and robots.

Programmable logic control devices are generalized in much the same manner as CNC machines.

Summary. Thus, with different but similar devices, generalization is possible. An abstract base class defines the interface and common functionality of the generalized device. For each different real device, a subclass is derived from the base class. An access to an object of an original device class is replaced by an access, via a reference with the polymorphic base class type, to an object of a subclass. The methods are dynamically bound (virtual functions in C++).

This kind of variability and this transformation are common in most domains where technical devices are used. A transformation can be done as described only when the original device classes, like Robot, may be modified and derived as a subclass (like Standardized Robot) from the hot-spot subsystem abstract base class.

SECOND TRANSFORMATION: DEVICE INDEPENDENCE

This transformation deals with both the variability of machines and devices and the variability of associations—that is, a device is exchanged for a completely different kind of device with dissimilar

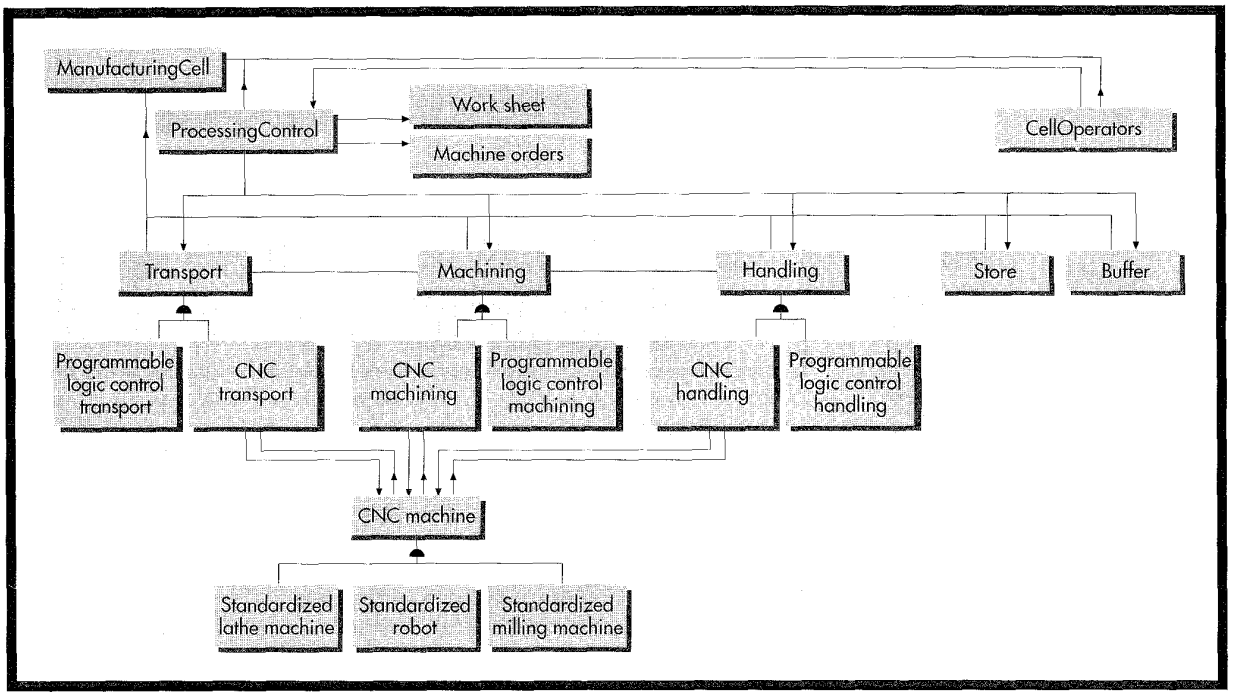


Figure 4. *Manufacturing cell with Machining, Transport, and Handling hot-spot subsystems.*

interfaces, and thus generalization isn't possible. How then can we attain device independence? For example, suppose a portal robot were replaced by two programmable logic control units: a conveyor belt for transport and a unit for handling. In this case, ProcessingControl would request, instead of a CNC machine service from the robot, two dissimilar programmable logic control services from two different units.

Which—not how. To make ProcessingControl reusable with respect to a change of transport system, you must incorporate knowledge of which transport service is to be done, rather than how the action is to be performed. Thus, you must abstract from the particulars of the device-related specific operations.

ProcessingControl should request transport services explicitly; for example, "move pallet from store position X to buffer store Y," and not implicitly via device-related service requests. To make this possible, we introduce a hot-spot subsystem for addressing transport. The abstract base class Transport defines the protocol for the transport services. For each (principally different) implementation of transport services by, for example, a robot or a conveyor belt, a concrete subclass is derived from the abstract Transport class.

The CNC Transport subclass contains a reference to the generalized device, such as the portal robot, that implements the transport. A subclass can be adapted to different cell configurations: It contains subclass-specific tables that describe the mapping from different transport tasks to robot control programs, for example, and from the abstract position parameter of the transport service to the respective parameters of the robot control program. CNC Transport transforms the transport service requests that it receives into robot service requests and forwards them to the referenced device.

By this mechanism, which is known as adapter design pattern,[10] we adapt a device to the application-specific transport functionality by deriving an adapter subclass from Transport. An adapter class is required because the device itself cannot be derived from Transport as a subclass. The reason is that a device, like a robot, "is not a" kind of transport; it can also be used for other manufacturing tasks, such as handling.

Exchanging the robot with a programmable logic-controlled conveyor belt, for example, has no effect on the ManufacturingCell and ProcessingControl classes. During the configuration of the ManufacturingCell object, a CNC Transport object could be replaced by a programmable logic control Transport object, and a Robot object by a ConveyorBelt object, without class modification or recompilation. The abstract manufacturing application logic is performed identically in all configurations.

As we did for transport, we introduce a hot-spot subsystem for handling and machining. Each is an abstract class with subclasses for CNC and programmable logic-control handling and machining. Since the abstract application logic of ProcessingControl should not require an implicit knowledge of which transport and handling devices were associated with a specific machine, the association of Handling and Transport objects to Machining objects is described explicitly with two relations, as shown in Figure 4.

Summary. When a domain has different configurations consisting of dissimilar devices used to perform the same task, you make the application logic independent of the actual device configuration by introducing a new level

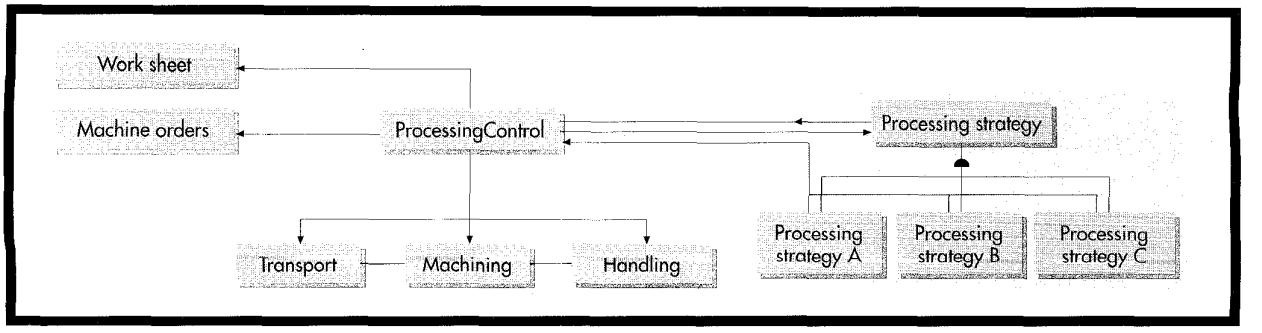


Figure 5. *ProcessingStrategy provides an easily modifiable application logic.*

of abstraction with one or more hot-spot subsystems. A base class defines services specific to an application-related task (in this case, transport from manufacturing). For this reason, we call these classes application object classes (also known as business object classes) which provide application-related services, thus abstracting from the (standardized) devices used in the application domain. Since the existing device classes cannot be changed, an application object class is mapped to the different device classes by an adapter design pattern[10]: For each device class used, a subclass is derived from the application object base class.

A subclass is parameterized with a table that describes how to map application-related service requests to device-specific service requests. You can tailor the framework according to the specific application configuration by providing specific table contents. An application object transforms a received request for a service, and passes it, possibly partitioned into several device service requests, to a standardized device object. For all configurations, ProcessingControl requests identical services from the application object layer. Thus, it has become reusable.

In many application domains the same task can be performed by means of different devices. To send a message to a coworker, for example, you could send it via phone, fax, or e-mail. Each device provides completely different services yet accomplishes the same result. In all application domains with this characteristic, the device independence transformation introducing application objects may be applied to reuse the application logic.

THIRD TRANSFORMATION: APPLICATION LOGIC EXTRACTION

This transformation addresses local configuration changes. They require slightly different versions of the part-processing application logic. For example, a part-processing sequence that calls for separate loading and unloading steps with a single-part gripper can combine loading and unloading into one step when a double-part gripper is used. The upshot is that such changes require a complete exchange of the ProcessingControl class.

Breaking it down. To simplify the effort in modifying the part-processing application logic, we divide the ProcessingControl class into several smaller, more specific classes[11] as shown in Figure 5. In doing so, we follow the strategy design pattern[10] and introduce a ProcessingStrategy class, which provides a ProcessParts service that performs the part-processing application logic. To accommodate different part-processing sequences, we define ProcessingStrategy as an abstract base class and derive different processing subclasses that overwrite ProcessParts. The part-processing sequence of a ProcessParts method requests a set of services like LoadPart or UnloadPart from a mediator class not shown in Figure 5 (these services are also generalized).[9] Thus, it is defined in a manner similar to that in which the abstract application logic is described.

Once again, we have introduced a hot-spot subsystem to handle application logic variations. When reading the next machine order and work sheet entry, ProcessingControl dynamically sets a polymorphic reference of the type ProcessingStrategy to reference the specific processing strategy to be used depending on the machine order equipment configuration. For detailed part processing, it calls the Process Parts method via this reference.

Summary. In this transformation, different variations of the application logic are performed in different manufacturing configurations. To make the application logic easy to modify, we extract it into a separate hot-spot subsystem. The application logic of a subclass strategy is composed of high-level service requests (provided by the base class or by a mediator class.[11]) This transformation can be applied to any domain where there are variations of the application logic.

OUTCOME

As Figure 6 shows, we introduced by three transformations three generalization layers into the class structure of OSEFA: a standardized machines and devices layer, an application objects layer, and an application logic layer. In these layers, OSEFA contains the following components:

- models of entities such as machines and robots;
- concepts for transport, handling, and machining; and
- the application logic that describes how a part is processed.

In addition to these principal aspects of framework design, we had several other design items to address before the framework could be implemented, such as

- central structuring[11] details;

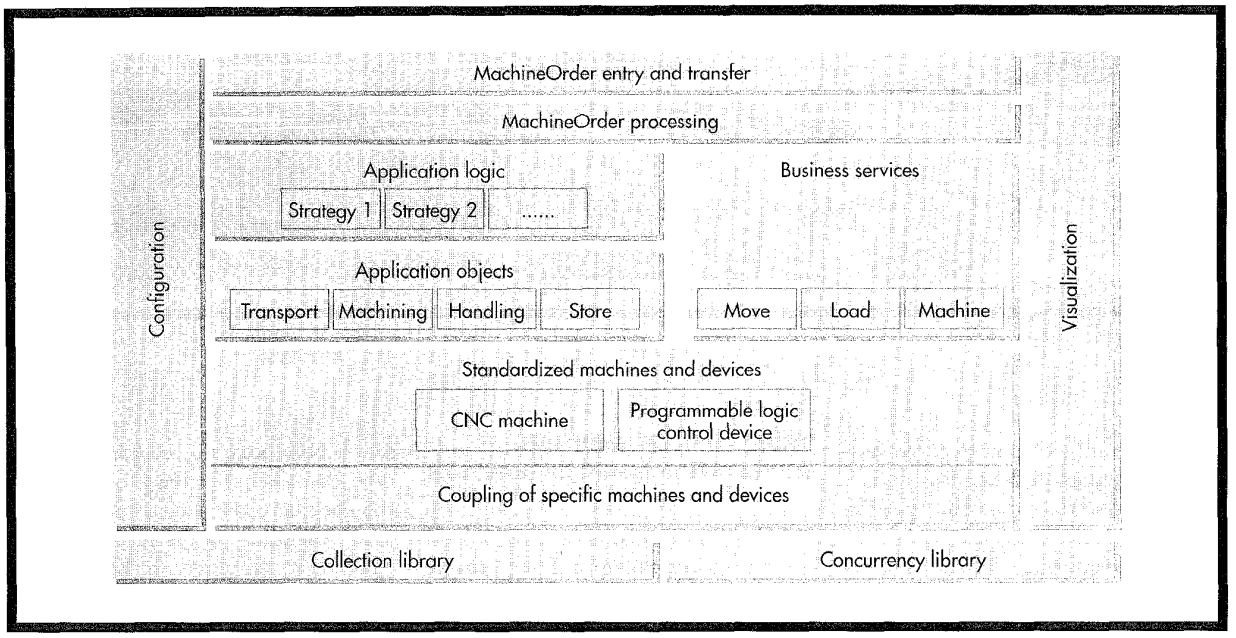


Figure 6. *Layer and subsystem structure of the OSEFA manufacturing framework.*

♦ concurrent execution of an optimized manufacturing logic with shared resources such as buffer stores;

♦ concurrent processing in a reactive system (similar to that described by Giuseppe Menga and his colleagues[12]);

♦ refinement of areas like store and buffer store, sometimes combined with additional functionality like the interactive shop-floor entry of machine order lists and worksheets;

♦ visualization of the manufacturing cell processing for any possible configuration; and

♦ configuration of an application from the framework.

Results. It takes roughly a day (a few days when we include testing) to create an application from the DSBB framework. Adding a few more days for the analysis of a manufacturing cell and time for complementing not-yet-existing framework components, the overall effort and elapsed time is still very small compared to the time it usually takes to develop the control software in an individual project, which ranges from a few person-months to person-years.

Before starting, our project group expected that a framework would reduce both cost and time to develop a domain-specific application by a factor of three to four. In fact, our actual results indicate that OSEFA may reduce cost and time by as much as a factor of 10.

Costs. Developing a DSBB framework for a specific domain requires considerable investment. Our DSBB framework required twice the effort typically spent developing a fixed domain application—and this figure only includes the costs of domain analysis and class structure generalization, not the time spent learning about framework structuring and design. This cost was reduced by initially providing only the components required for the first application. When creating more applications, you can incrementally increase the number of application components. The effort will vary depending upon domain.

Our experience strongly indicates that developing a framework is worth the effort. The initial investment will generally be returned after you develop the second or third application. There are also significant testing advantages and a substantially decreased application development time.

In a first test stage, we used the framework configuration features to replace all real machines and devices with simulated machine and device classes, and then used visualization to see that the system ran correctly. Thus, we could find logic errors with relatively little effort, gradually adding real classes, then real machines, in subsequent test phases. We thus saved considerable testing effort and cost.

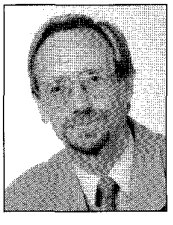
As to application development, we had very little time to modify our cell configuration before a scheduled demonstration. We had to produce a new product with completely different parts; as a result, we had to add a laser scanner as a measurement station so that the cell had two machining stations (measurement and the existing lathe) and introduce new part-processing sequences to produce the new products. It took less than a week to write the new processing strategy classes and configure the new application from the framework—less time overall than it took to make mechanical and equipment changes. Meeting such a short time-to-market deadline would have been impossible without the framework. In making these changes, we were confronted with unplanned variability which we could not have accommodated if not for OSEFA's sufficiently general architecture.

ACKNOWLEDGMENTS

My thanks to Clemens Ballarin, Franco Indolfo, Frank Mueller, and Jochen Peters who worked with us to build OSEFA and to the German Research Council (Deutsche Forschungsgemeinschaft) for partial support of the project.

REFERENCES

1. M.D. McIlroy, "Mass Produced Software Components," in *Proc. Nato Conf. Software Eng.*, Garmisch-Partenkirchen, NATO Sciecne Committee, NATO, Brussells, 1969.
2. G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings Redwood City, Calif., 1991.
3. P. Coad and E. Yourdon, *Object-Oriented Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1991.
4. R.E. Johnson and B. Foote, "Designing Reusable Classes," *J. Object-Oriented Programming*, June 1988, pp. 22-35.
5. R.E. Johnson and V. Russo, "Reusing Object-Oriented Design," Tech. Report UIUCCDS 91-1696, Dept. of Computer Science, Univ. of Illinois, Urbana, Ill., 1991.
6. A. Weinand, E. Gamma, and R. Marty, "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework," *Structured Programming*, June 1989, pp. 63-87.
7. J. Brant and R.E. Johnson, "Creating Tools in HotDraw by Composition," in *Technology of Object-Oriented Languages and Systems TOOLS 13*, B. Magnusson et al., eds., Prentice-Hall, Englewood Cliffs, N.J., 1994, pp. 445-454.
8. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, Mass., 1994.
9. H.A. Schmid, "Design Patterns for Constructing the Hot Spots of a Manufacturing Framework," *J. Object-Oriented Programming*, June 1996, pp. 25-37.
10. E. Gamma et al., *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, Reading, Mass., 1994.
11. H.A. Schmid, "Creating the Architecture ofa Manufacturing Framework by Design Patterns," *Proc. OOPSLA '95*, ACM, New York, 1995, pp. 370-384.
12. A. Aarsten, G. Elia, and G. Menga, "G++: A Pattern Language for Computer Integrated Manufacturing," *Proc. PLOPS '94*, J. Coplien and D. Schmidt, eds., Addison-Wesley, Reading, Mass., 1995, pp. 91-118.
13. *Domain Analysis and Software Systems Modelling*, R. Prieto-Diaz and G. Arango, eds., IEEE CS Press, Los Alamitos, Calif., 1991.



Hans Albrecht Schmid is a professor of computer science at the Fachhochschule Konstanz, where his interests are in software engineering, OO development and methods, emphasizing frameworks and design patterns, and real-time system development. Before joining FH Konstanz, he spent 10 years with the IBM Development Laboratory in Boeblingen.

Schmid received an MS in electrical engineering from the Technical University, Stuttgart, a diploma in computer science from the Institute Nationale Polytechnique de Grenoble, and a PhD in computer science from the University of Karlsruhe.

Address questions about this article to Schmid at Fachbereich Informatik, Fachhochschule Konstanz, Brauneggerstrasse 55, D-78462 Konstanz, Germany; phone x-49-7531-9836-39 or -12; fax, -13; schmidha@rz-uxazs.fh-konstanz.de.