SURVI

The state of object-oriented design is evolving rapidly. This survey describes what are currently thought to be the key ideas. Although it is necessarily incomplete, it contains both academic and industrial efforts and describes work in both the United States and Europe. It ignores well-known ideas, like that of Coad [6] and Meyer [34], in favor of less widely known projects.

Research in object-oriented design can be divided many ways. Some research is focused on describing a design process. Some is focused on finding rules for good designs. A third approach is to build tools to support design. Most of the research described in this article does all three.

We first present work from Alan Snyder at Hewlett-Packard on developing a common framework for object-oriented terminology. The goal of this effort is to develop and communicate a corporate-wide common language for specifying and communicating about objects.

We next look into another research activity at Hewlett-Packard, led by Dennis de Champeaux. De Champeaux is developing a model for object-based analysis. His current research focuses on the use of a trigger-based model for inter-object communications and development of a top-down approach to analysis using ensembles.

We then survey two research activities that prescribe the design process. Rebecca Wirfs-Brock from Tektronix has been developing an object-oriented design method that focuses on object responsibilities and collaborations. The method includes graphical tools for improving encapsulation and understanding patterns of object communication. Trygve Reenskaug at the Senter for Industriforskning in Oslo, Norway has been developing an object-oriented design method that focuses on roles, synthesis, and structuring. The method, called Object-Oriented Role Analysis, Syntheses and Structuring, is based on first modeling small subproblems, and then combining small models into larger ones in a controlled manner using both inheritance (synthesis) and run-time binding (structuring).

We then present investigations by Ralph Johnson at the University of Illinois at Urbana-Champaign into object-oriented frameworks and the reuse of large-scale designs. A framework is a high-level design or application architecture and consists of a suite of classes that are specifically designed to be refined and used as a group. Past work has focused on describing frameworks and how they are developed. Current work includes the design of tools to make it easier to design frameworks.

Finally, we present some results from the research group in object-oriented software engineering at Northeastern University, led by Karl Lieberherr. They have been working on object-oriented Computer Assisted Software Engineering (CASE) technology, called the Demeter system, which generates language-specific class definitions from language-independent class dictionaries. The Demeter system includes tools for checking design rules and for implementing a design.

Common Terminology

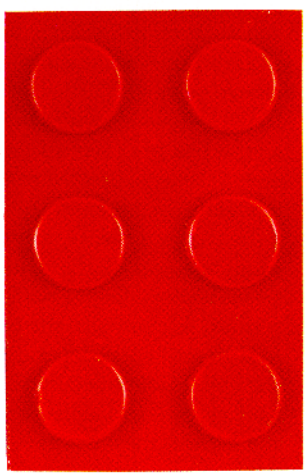
Hewlett-Packard is involved in many activities that involve object-oriented technology. In addition to object-oriented programming [4, 15, 52], these activities include object-oriented databases [13], user interfaces [18], application architectures [19], application integration platforms [39], distributed systems [2], and network management architectures [20]. Each area has an associated external technical community, and many have associated standards activities. These areas differ in both the forms in which object-oriented concepts appear and the terms used to refer to the concepts. This diversity has caused serious communication problems and has hampered the convergence of these technologies to provide comprehensive and consistent object-based solutions.

To address this issue, representatives from the various technologies were brought together to exchange information and develop a shared understanding. The first step in developing the glossary of common terminology was to identify and define a set of core concepts that could be used to characterize and contrast the various object-oriented technologies familiar to the task force mem-

EYING

CURRENT RESEARCH IN OBJECT-ORIENTED DESIGN

Rebecca J. Wirfs-Brock
and Ralph E. Johnson



bers. These concepts were defined in a general way to apply to the wide variety of technologies under consideration. Each core concept that was identified had a term selected for it. The result was the abstraction of the essential common core concepts of object-oriented technology, and a common terminology. This common terminology has been endorsed by the management of the computer business within Hewlett-Packard and is being promoted within corporate training programs.

The essential concepts are summarized as follows:

- An *object* embodies an abstraction. It provides *services* to its *clients.* This principle emphasizes that an object is not just a collection of data. It explicitly embodies an abstraction that is meaningful to its clients, which may be programs or users. The services are computations that are appropriate to the abstraction.
- Clients *request* services from objects. Clients respect the abstractions embodied in an object. Objects are *encapsulated:* clients are prevented from making direct access to the data associated with an object. Instead, clients issue requests for services that are performed by objects. Performing a request involves executing some code, a *method,* on the associated data. A request identifies the requested service (the *operation*), as well as the objects that are to perform the service. These objects can be identified unambiguously and reliably (this is called *object reference*). Requests may include other parameters, and may return results. The set of services that an object provides may be described in the form of an *interface description.*
- Requests can be *generic.* A client can issue the same request to different kinds of objects that provide similar services. Specifically, performing a request may involve the execution of different code depending upon the objects identified in the request.
- Objects can be classified in terms of the services they provide. This classification may be based on explicit interface descriptions. An object could provide a subset of the services provided by another object, leading to *hierarchical classification* and an *interface hierarchy.*
- Objects can share implementations. Mechanisms are generally provided that allow multiple objects, called *instances,* to share the same implementation (often called a *class*). Also, mechanisms are often provided by which the implementation of one object cannot just share the implementation of another object, but can also extend or refine it (*implementation inheritance* or *delegation*) [48].

There are several commonly used terms that have been frequent sources of confusion and miscommunication within Hewlett-Packard. The term that causes the most confusion is *encapsulation.* Encapsulation has three possible meanings: the enforcement of abstraction barriers; the act of integrating foreign components into a system; and the mechanism for controlling access to services by different users. (The recommended terms identified by the task force for these three concepts are *encapsulation, embedding,* and *protection.*) Another confusing term is *inheritance.* Inheritance has two primary meanings: a mechanism by which object implementations can be organized to share descriptions; and a classification of objects based on common behavior or common external interfaces. (The recommended terms for these concepts are *implementation inheritance* and *interface hierarchy.*) Other confusing terms are *type* and *class,* whose multiple meanings refer to either the external interfaces of objects or the implementations of objects.

Snyder has found that the distinctions between multiple meanings can be subtle, even to people who are familiar with basic concepts. The results of the common object terminology effort are available in two technical reports [49, 51]. In addition to providing a definition for each concept, the reports identify synonyms, give examples, state the importance of the concept, and present the rationale for the choosing the recommended term.

Rather than develop a comprehensive terminology, the most important concepts and those with multiple meanings or ambiguous terms were identified and defined. The terms were selected to be broadly applicable to multiple domains, and were not restricted to programming terminology. This led researchers at Hewlett-Packard to adopt, in some cases, terminology that is different from the more commonly used object-oriented programming terminology. Adoption of the terminology has been the result of personal initiative and leadership by members of the original task force. For example, the terminology has been adopted in the architectural documents for Hewlett-Packard's NewWave Computing Architecture, the company's strategic initiative for its computer business.

We present the entry from the HP technical report for the term *generic request* to illustrate the complete descriptions that were developed.

Generic Request

Definition:

A *request* is a statement that specifies a service to be carried out by objects. A request has a name, identifies the objects that are to provide the service (the providers), and may take arguments and produce results. A *generic requrest* is a request that may be issued to different objects that provide (similar) services with different implementations and possibly different behaviors. The request itself does not determine how the services will be performed. When a request is issued, a selection process determines the actual code to be executed to perform the service. More than one object can participate in providing a service in response to a request.

Example:

A print request can be made to any printable object (e.g., a document or spreadsheet). The request may also specify a device object where the document will be printed.

Importance:

Generic requests are a major factor in the reusability of object-oriented programs. Code written in terms of generic requests can be used for different purposes when the requests are sent to objects that interpret them differently. In object-oriented user interfaces, generic requests allow multiple applications to share a common interaction style, improving ease of use.

Synonyms and Related Terms:

In the Iris database, issuing a generic request is called *function invocation.* In C++ it is called *virtual member function invocation.* In general, a request may designate multiple objects to provide the service. A *message* is a generic request for a service issued to a single object; issuing such a request is called message sending. The ability to support generic requests is also called *polymorphism* and *function overloading.*

Rationale:

The use of the word *generic* highlights the feature that a single request may denote a range of related services. We de-emphasize the traditional term message for two reasons: One is the common misconception that message sending implies concurrent execution by the sender and the receiver. The other is the implication that a message is sent to a single location at which it is handled. Although in traditional object-oriented systems, services are provided by individual objects, systems like the Iris database and the Common Lisp Object System (CLOS) have begun to explore more general models in which the implementation of a single service is provided jointly by multiple objects. While the integration of this concept with the traditional object-oriented model is not yet clear, it is clear that the more general model is needed to handle certain real problems. A classic example is the problem of printing a document on a printer, the implementation of which may differ based on both the kind of document and the kind of printer.

Common terminology within a single organization is only an intermediate solution. The ultimate goal is consensus within the technical community at large. As a step in this direction, Hewlett-Packard is participating in the Object Management Group (OMG), an industry consortium chartered to promote the widespread adoption of object technology. The OMG is actively working to influence the future directions of object-oriented technology, specifically through the adoption of a platform-independent object-oriented applications environment. The work described above has been incorporated and extended by the OMG technical committee in creating a "standards manual" to guide the formulation of and responses to requests for technology.

An "abstract object model" [50] has been defined which provides an organized and more detailed presentation of concepts and terminology. This abstract object model also partially defines a model of computation. The partial nature of this description is in keeping with the OMG's policy of adopting existing technologies rather than designing new ones. Any existing technology submitted to the OMG will likely define its own concrete object model based upon this abstract object model. The abstract object model provides a framework for such concrete object models. A concrete object model would elaborate upon the abstract object model by making it more specific, for example, by defining the form of a request, and would populate the abstract object model by introducing specific instances of object model entities, such as specific operations.

To illustrate the evolution of terminology, here is the definition for request from the abstract object model:

> Clients request services by issuing *requests.* A request is an event (i.e., something that occurs at a particular time during the execution of the computational system). The information associated with a request consists of an *operation* and zero or more (actual) parameters. Operations are (potentially) *generic,* meaning that a single operation can be requested of objects with different implementations, resulting in observably different behavior. Operations are created by explicit action; each such action creates an operation that is distinct from operations created previously or in the future. A *value* is anything that is a possible (actual) parameter in a request. A value may identify an object, for the purpose of performing the request. A value that identifies an object is called an *object name.* A *handle* is an object name that unambiguously identifies a particular object. Within certain pragmatic limits of space and time, a handle will reliably identify the same object each time the handle is used in a request. A request causes a service to be performed on behalf of the client. One outcome of performing a service may be that some results are returned to the client. The results associated with a request may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

There is a subtle change in this new description from the original. In the earlier definition the *request* was called generic. In the newer terminology, it is the *operation* that is called generic. This change resulted from making the definition of request more formal. Several possible meanings were considered: the form issued by the user (for example, an invocation form in a program text), the information provided (the operation and the actual arguments), or the computational event itself.

The last option was chosen both for its utility, since the results are associated with the event, and ease of formalization, since the syntactic form cannot easily be formalized in an abstract form. The original definition of generic request assumed the first meaning. With the new meaning, it no longer made sense: the same request (event) cannot be is-

sued to different objects. Therefore, the concept of generic was associated with operation.

Efforts are continuing within Hewlett-Packard and elsewhere to further refine these concepts and the abstract object model and to work toward consensus within the technical community.

Object-Oriented Analysis

Another research activity at Hewlett-Packard addresses the object-oriented paradigm for analysis. The goal of this research is to develop an analysis method that can be integrated with object-oriented design. A primary objective of this research is to develop a method that does not assume sequential computation. [9].

The analysis method should allow for what de Champeaux terms *unlimited formalization.* The method should not impose formalization on the analyst. However, if validation of the implementation is required, it should be verifiable against the results of analysis.

The object-oriented paradigm classically has its roots in sequential programming languages. Object interaction in such a context is too simplistic: the sender passes an operation name and arguments to the receiver. Control is initially passed to the receiver. The receiver next executes the desired operation and sends the result back to the sender. Finally, control is returned to the sender. This model of control and information flow is not rich enough to describe all the causal connections between objects an analyst needs to model.

Shlaer and Mellor have developed an object-oriented process model that relies on data flow diagrams from Structured Analysis to describe the actions in their state models [47]. In their model, interaction between objects is described indirectly via the occurrence of an external data store in a data flow diagram.

De Champeaux is exploring whether triggers provide a more direct mechanism for modeling causal interactions between objects. A trigger does not carry data, the initiator is not suspended, and it does not expect a return value. The only effect of a trigger is to initiate a state change by the recipient.

Another subtlety that must be modeled is how to deal with a trigger that cannot be handled by the recipient, perhaps because an additional condition for a triggered transition is not satisfied. Should the trigger be lost, buffered, or signal an error condition? Each of these responses is appropriate under certain circumstances. This suggests that a richer interobject interaction model than a trigger is necessary.

This has led to consideration of additional object interaction forms, such as:

trigger-and-wait-for-acknowledgement (where the initiator waits for acknowledgement of receipt of the trigger),

send-no-wait (where data and the trigger are simultaneously transmitted),

send-and-wait-to-acknowledge-of-reception (where the initiator triggers a transition, while transmitting a value that will be consumed by an action on the transition; the sender blocks until it receives acknowledgement that the trigger and value have arrived), or

send-and-wait-for-reply (similar to the above example except for the sender is blocked until a value is returned).

Determining an appropriate set of additional forms to describe interobject interactions is a current research topic.

Ensembles

An analyst using traditional structured analysis techniques obtains a top-down view of a system. Process decomposition is a well-known technique. Similar mechanisms are needed for objects.

For example, in analyzing a banking application, an interest rate, a branch office, a teller machine, a corporate account, a loan officer, or a monthly statement are all candidate objects to model during analysis. However, they obviously represent different layers in the problem domain.

De Champeaux is investigating an appropriate abstraction layering and decomposition technique for selecting objects during analysis that facilitates such a layered analysis. Currently he is exploring ensembles as a technique for creating and analyzing objects in an ordered fashion.

Ensembles represent a cluster or bundle of less abstract entities which are either objects or lower-level objects. Ensembles, like objects, can be modeled by attributes and optionally a state-transition machine, can interact with other objects or ensembles, and have an interface model. A major distinction between ensembles and objects is that an ensemble has internal parallelism, while an object is a finite state machine.

The main purpose of an ensemble is to hide details of a set of objects or subensembles that are irrelevant outside the ensemble. Like classes of objects, classes of ensembles can be modeled. An important part of an ensemble's information model is a description of its constituent objects and subensembles. Additional ensemble attributes may model features that apply to an ensemble's constituents as a whole. For example, consider a fleet of ships represented as an ensemble. The individual ships share the direction in which they are going. Thus, we can model direction as an attribute of a fleet. Summary information may also be modeled such as the number of ships in the fleet.

When an ensemble has nonconstituent attributes, it is often appropriate to develop a state-transition model for it. Inter-ensemble or object interactions can then be described. A major difference between an object and an ensemble is that ensembles have a forwarding mechanism for triggers and messages that mediates between external entities and ensemble constituents. From outside an ensemble, it may appear as if messages to an ensemble directly cause ensemble constituents to change state. For example, the return-to-port transition causes the direction of all ships in the fleet to change. When we

look inside the fleet ensemble, we see a different triggering and messaging pattern that actually achieves these consequences. Introducing ensembles thus allows low-level mechanisms to be hidden from higher-order functionality.

Research in Responsibility Driven Design

Over a period of six years, Tektronix developed one of the largest and most experienced groups of Smalltalk programmers and produced several major commercial and internal applications [12, 35, 57]. Much experience was gained in the process. In the last few years a number of individuals have focused on developing and teaching a method for designing object-oriented applications [59]. Past experience in Smalltalk development led to a strong sense of what constitutes good design. The result is a design process that has been applied to a number of small and medium-sized engineering endeavors at Tektronix and elsewhere [60].

In developing a method for object-oriented design the following goals were set:

1. Develop a model that encourages exploration of alternatives early in the design process, and that provides a structure for analyzing and improving initial design decisions.
2. Develop simple tools that help a design team to reason about a design. It should be easy to record and modify design decisions.
3. Develop language-independent methods and guidelines.

Responsibility Driven Design

Responsibility-driven design models an application as a collection of objects that collaborate to discharge their responsibilities. Responsibilities are a way to apportion work among objects that comprise the application. This approach stresses focusing on what actions must be accomplished and which objects will accomplish them. How each action is accomplished is deferred until after a model of objects and the interactions is created and understood.

Responsibilities include two key items:

- the knowledge an object maintains, and
- the actions an object can perform.

Responsibilities are meant to convey a sense of the purpose of an object and its place in an application. Responsibilities represent the publicly available services defined by objects. Note that responsibilities have been defined for objects, not classes, though the responsibility of a class can be defined as the responsibilities of its instances.

Focusing on the responsibilities of objects maximizes information hiding and encapsulation. Information-hiding distinguishes the *ability* to perform some act from the specific steps taken to do so. An object reveals its abilities publicly, but it does not tell how it knows or does them. An object may need to know and do other things in order to fulfill its public responsibilities, but those things are considered private to the object.

The responsibilities of an object are all the services it provides for all objects that communicate with it. Objects fulfill their responsibilities in one of two ways: by performing the necessary computation themselves, or by collaborating with other objects.

Exploration

The process of design can be partitioned into two distinct phases, as shown in Figure 1. To start, object-oriented design is exploratory. The designer looks for classes of objects, trying out a variety of schemes in order to discover the most natural and reasonable way to abstract the system. During the initial exploratory phase of design the *primary* concern is to build a model of the key classes that will fulfill the overall design objectives. In this phase the major tasks are to

- discover the classes required to model the application,
- determine what behavior the system is responsible for, and assign

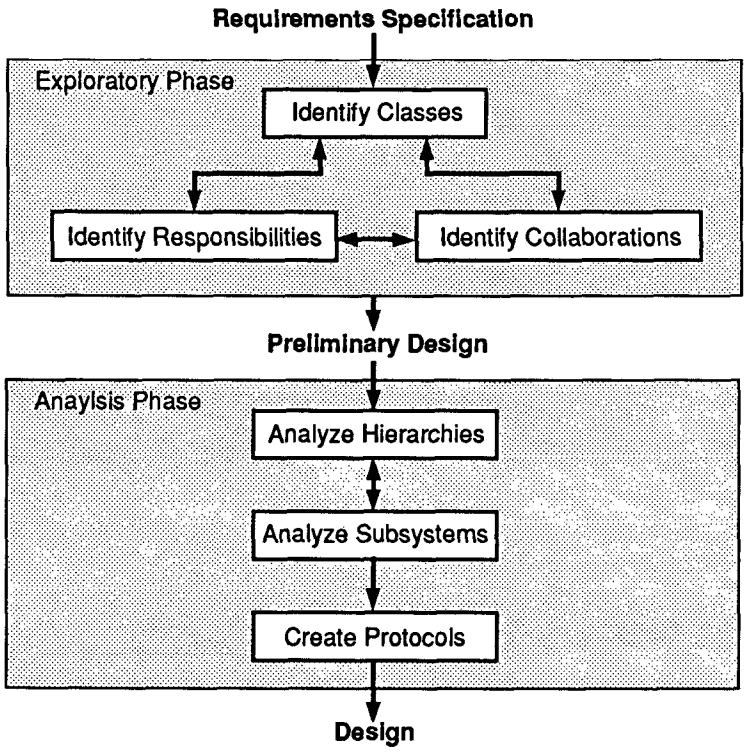


FIGURE 1. Phases of design: exploration and analysis.

these responsibilities to specific classes, and
- determine what collaborations must occur between classes of objects to fulfill those responsibilities.

Modeling is the process by which the logical objects in a problem space are mapped to the actual objects in a program. These steps produce a set of candidate classes for an application, a description of the knowledge and operations for which each class is responsible, and a description of collaborations between classes (i.e., between instances of those classes).

Recording The Initial Design

Beck and Cunningham [3] have found that index cards are a simple tool for teaching object-oriented concepts to designers. The responsibility-driven design method uses index cards to capture initial classes, responsibilities and collaborations. They also record subclass-superclass relationships and common responsibilities defined by superclasses.

Index cards work well because they are compact, easy to manipulate, and easy to modify or discard. Index cards can be easily arranged on a tabletop and a reasonable number of them viewed at the same time. They can be picked up, reorganized, and laid out in a new arrangement to amplify a fresh insight.

Each candidate class is written on an index card, as shown in Figure 2. Each identified responsibility is succinctly written on the left side of the card. If collaborations are required to fulfill a responsibility, the name of each class that provides necessary services is recorded to the right of the responsibility. Services defined by a class of objects include those listed on its index card, plus the responsibilities inherited from its superclasses.

Improving the Initial Design

Once an initial model has been constructed, it is crucial to turn a critic's eye on the design.

Without such attention, it is difficult to obtain the reusability and refinability benefits touted by object-oriented technology. It is particularly important to construct properly structured hierarchies, to identify abstract classes, and to simplify inter-object communications. During this second, highly analytical phase of design the *primary* activities are to

- factor the responsibilities into hierarchies to get maximum reusability from class designs,
- model the collaborations between objects in more detail to better encapsulate subsystems of objects, and
- determine the protocols and complete a specification of classes, subsystems of classes, and client-server contracts.

Paying careful attention to structuring abstract and concrete classes, first, *before* improving object collaborations, reduces rework required during later stages.

Factoring Hierarchies

A design is most extensible when a class inherits from another class only if it supports all of the responsibilities defined by that other class. Inheritance should model "is-kind-of" relationships: every class should be a specific kind of its superclasses [17, 26]. Subclasses that support all of the responsibilities defined by their superclasses are more reusable because it is easier to see where a new class should be placed within an existing hierarchy. A corollary of this principle is that if a set of classes all support a common responsibility, they should inherit it from a common superclass.

An important distinction to be made when designing a class is the primary purpose of abstract and concrete classes. Abstract classes are designed to be inherited. They exist solely to specify behavior that is reused by inheritance. Instances of abstract classes are never created as the system executes. Concrete classes are designed to be instantiated. Although it is often useful to inherit from a concrete class, concrete classes are usually not designed to be reusable by inheritance, but as components.

One way to factor responsibilities higher in a class hierarchy is to design as many abstract classes as possible. In general, the more concrete subclasses of an abstract class, the more likely the abstraction is to stand the tests of time and software enhancements. Only one responsibility is needed to define an abstract superclass, but at least two specific subclasses of it are required before one can hope to design a generally useful abstraction. Defining many abstract superclasses as possible means that much common behavior has been factored into reusable abstractions.

Tools for Understanding Object Interactions

Analyzing an exploratory design requires global understanding. Both graphical and conceptual tools are used to gain that understanding.

Contracts

A *contract* is a set of related responsibilities defined by a class. It describes the ways in which a given client can interact with a server. A contract is a list of requests that a client can make of a server. Both must fulfill the contract: the client by making only those

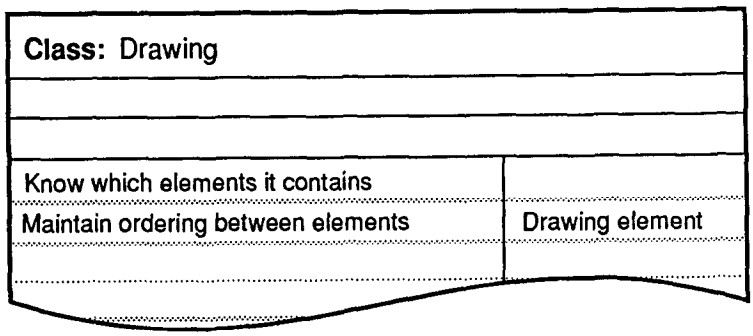


FIGURE 2. An Index Card with Collaborations

requests that the contract specifies, and the server by responding appropriately to those requests. The relationship is shown in Figure 3.

Responsibilities found in the exploratory phase are the basis for determining the contracts supported by a class. Not all responsibilities will be part of a contract. Some responsibilities represent behavior a class must have to support the fulfillment of contracts but which are not directly exposed to other objects. These are *private responsibilities.*

A class can support one or more distinct contracts. The word "contract" is not just another name for a responsibility. A responsibility is something one object does for other objects, either performing some action or responding with some information. A contract defines a cohesive set of responsibilities that a client can depend on. The cohesion between responsibilities is a measure of how closely those responsibilities relate to one another.

For example, all classes of numbers support a contract to perform arithmetic operations. That contract includes responsibilities to perform addition, subtraction, multiplication and division. For example, let us say a new class defines the responsibility for its instances to know how to add themselves to other instances of the class. These new objects cannot be used as servers in places where some type of number is expected. The new class defines the addition responsibility, but it does not support the entire set of responsibilities defined by the arithmetic contract.

Often a class supports only a single contract. However, when a class has multiple roles or when its services can be factored into sets that are used by distinct clients, it will support multiple contracts.

Subsystems of Classes

An application is composed of more than just classes. A complex system requires many levels of abstraction, one nested within the other. Classes are a way of partitioning and structuring an application for reuse. But a design often has groups of classes that collaborate to fulfill a larger purpose.

A *subsystem* is a set of such classes (and possibly other subsystems) collaborating to fulfill a common set of responsibilities. Although subsystems are not directly supported by existing object-oriented languages, they are an important way of thinking about large object-oriented systems. One way to test if a group of classes form a subsystem is to try to name the group. If the group can be named, the larger role they cooperate to fulfill has been named. A subsystem is not just a bunch of classes, it should form a good abstraction.

Collaborations Graphs

A *collaborations graph* helps analyze paths of communications and identify potential subsystems. It graphically displays the collaborations between classes and subsystems. The graph can be used to identify areas of unnecessary complexity, duplication, or places where encapsulation is violated. Collaborations graphs represent classes, contracts, and collaborations. In addition, collaborations graphs show superclass-subclass relationships.

A subclass in a responsibility-driven design should support all the contracts defined by its superclass. Therefore, in a collaborations graph, a superclass represents the contracts supported by all of its subclasses. This idea is represented by graphically nesting subclasses within the bounds of their superclasses.

One example of a subsystem, shown in Figure 4, is the printing subsystem encapsulating the classes Print Server, Printer, and its subclasses Dot Matrix Printer and Laser Printer. Together, these classes can be viewed as collaborating to print files. Although the Print Server collaborates with Queue, Queue is not part of the Printing Subsystem, because instances of the class Queue are used by classes outside the Printing Subsystem. A class is part of a subsystem only if it exists solely to fulfill the goals of that subsystem.

Subsystems simplify a design. A large application is made less complex by identifying subsystems within it and treating those subsystems as classes. An application can be decomposed into subsystems, and those subsystems can in turn be modeled until all required richness and detail have been specified. Ultimately, software is composed of classes, but to ignore the possibility of subsystems is to ignore one of the most fruitful aspects of the structure of an application.

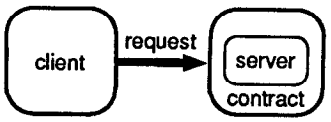


FIGURE 3. The Client-Server Contract

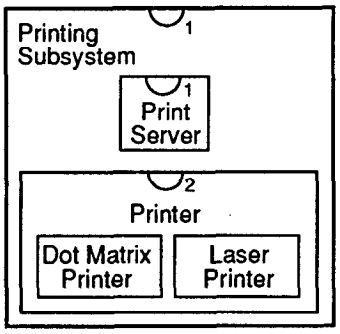


FIGURE 4. The Printing Subsystem Collaborations Graph.
Contracts are indicated by semicircles. Classes are drawn as rectangles, with subclasses nested within superclasses. An arrow is drawn from the client to the server supporting the contract. It is typical for many classes in a design to support just one contract, and, in fact, to inherit it from a superclass.

Subsystems are only conceptual entities; they do not exist during execution. They therefore cannot directly fulfill any of their contracts. Instead, subsystems delegate each contract to a class within them that actually supports the contract.

Because clients use the functionality of a subsystem through a clearly defined set of contracts, subsystem functionality can be extended without disrupting the rest of the application. A new contract can be defined, or an existing contract can be extended to provide access to the addi-

tional functionality. For example, we could extend the Printing Subsystem by adding the ability to print at a specified time or to print a specified number of copies. Existing contracts would adequately deal with the new functionality; the Printing Subsystem would still print the contents of a file (the old contract), but would do so in different ways (the new functionality).

Guidelines for Simplifying Interactions

Subsystems are identified in order to simplify the patterns of collaboration. Without such simplification, the communication paths could flow from nearly any class to any other, with only the slenderest of justifications and no coherent structuring. Such anarchic flow leads to spaghetti code—the same problem that structured programming was designed to avoid. The problem is evident when one looks at a collaborations graph for such an application. The graph itself looks like spaghetti; it cannot be understood, and the application it represents is consequently impossible to maintain or modify sensibly.

Simplifying the patterns of collaboration translates into a simplification of the collaborations graph. Places where the graph is complex are areas that likely need to have collaborations simplified. Often collaborations graphs are drawn repeatedly to test simplification alternatives.

Basic guidelines are used to simplify patterns of collaboration for the following purposes:

- Minimize the number of collaborations a class has with other classes or subsystems.
- Minimize the number of classes and subsystems to which a subsystem delegates. Another way of stating this principle is that the classes within a subsystem should be encapsulated whenever possible.
- Minimize the number of different contracts supported by a class or a subsystem.

Implementing Abstract Classes

Abstract classes are an important part of an object-oriented design because they not only define behavior that is shared by many classes, they provide a reusable design for their subclasses. An implementation of an abstract class will use three kinds of methods to describe the contract between subclass and superclass, and between superclass and subclass. These are termed base methods, abstract methods and template methods.

Base methods provide behavior that is generally useful to subclasses. The purpose of base methods is to implement in one place behavior that can be inherited by subclasses.

Abstract methods provide default behavior that subclasses are expected to override. The behavior does not do anything particularly useful, and subclasses are expected to reimplement the entire method. The purpose of abstract methods is to fully specify the subclasses responsibilities. Thus, the designer of a subclass uses the abstract methods as a specification.

For example, the abstract class Displayable Object might define the method display as an abstract method. The method might, perhaps, display a black box the size of the object's bounding box. In order for any element to display itself accurately, all subclasses of Displayable Object must reimplement the method display to provide accurate, reasonable display behavior for the particular kind of Displayable Object.

Template methods provide step-by-step algorithms. Each step can invoke an abstract method, which the subclass must define, or a base method. The purpose of a template method is to provide an abstract definition of an algorithm. The subclass must implement specific behavior to provide the services required by the algorithm.

For example, the abstract class Filled Element, a subclass of Displayable Object, might define the method display as a template method with this algorithm:

```
drawBorder
drawInterior
```

This alters responsibilities of its subclasses from the abstract operation specified in Displayable Object. Each subclass of Filled Element must implement the methods drawBorder and drawInterior in such a manner that they provide reasonable behavior.

An abstract class and its methods therefore serve as a minimal specification of each of its subclasses. An important part of specifying an abstract class is specifying the behavior for each method that is inherited by its subclasses. Specification of methods for an abstract class should state whether the method is an abstract method that must be overridden, or a base or template method that should be directly inherited.

Defining Class Structure

The implementation of a class hierarchy should push details about a class's structure as low as possible in the hierarchy. Subclasses can override inherited behavior, but not structure, so it is better to delay design decisions about structure as long as possible.

A responsibility is a statement of intent. It is general; it says nothing of *how* a responsibility is supported—details of structure or algorithms. If a superclass supports its responsibilities in the most generic way possible, there will not be any implementation details to impede a creation of a new subclass that wishes to inherit its responsibilities. Each subclass is free to implement the responsibilities in a way most appropriate for it.

Abstract classes define default implementations for some methods in order to make it easier to create subclasses. If they must depend on implementation details, those details should be accessed by sending a message to the object itself [58]. Messages sent to the object can easily be overridden by subclasses, allowing subclasses to provide a mapping from the abstract implementation assumed by the superclass to

the concrete implementation they support.

For example, consider an abstract class **Point** with two concrete subclasses, **Cartesian Point** and **Polar Point.** The addition of two points can be abstractly defined in terms of adding the x and y coordinates. If the x and y coordinates are accessed through a message send, each subclass can then supply its own implementation of these messages based on its internal representation.

Object-Oriented Software Engineering

For the past 10 years, the group at Senter for Industriforskning (SI) has been developing highly interactive, flexible and personalized work environments for executives and other professionals in public service, commerce and industry. The power of object orientation has been critical to their success. All their efforts in software engineering have been aimed at providing improved leverage for their development efforts.

When the group started development work in 1983, there was a conflict between functional specifications (which clearly indicated that Smalltalk-80 was the preferred development environment), and the requirements for reliability and maintainability (which clearly indicated a well-proven software engineering environment based on a traditional programming language). The group at SI opted for the Smalltalk-80 environment because of its object orientation, development environment and rich class library. Over the years they have developed a personal work environment based on Smalltalk to augment the initial system.

This personal work environment consists of a Smalltalk image containing a kernel module and a number of optional function modules that can be configured to suit an individual user's requirements. There are also a number of background services such as a persistent object store, that are mainly written in C. There are approximately 100,000 lines of Smalltalk-80 source code.

The group at SI believes that the key to program quality is simplicity: simple models, simple designs, simple code. They also believe that if a problem is really understood, a simple solution can be found. Their strategy has been to develop methods and tools that first permit the modeling of small subproblems until they are fully understood, then to combine the small models into larger ones in a controlled manner.

They term their method OORASS, Object-Oriented Role Analysis, Synthesis and Structuring, because of its three critical operations [41]. These operations are based on the encapsulation, inheritance and dynamic binding properties of object orientation. Analysis describes subproblems by encapsulating behavior in the objects of an object model, which is termed a *Role Model.* Synthesis defines composite objects by inheriting behavior from several simpler objects. Structure Specification prescribes how objects can be bound together in an actual instance of a system.

The goal of the OORASS research is to help people create an organized structure of collaborating objects and to represent such a structure in a computer. They believe that three distinct abstractions of objects are needed if the full benefits of object orientation are to be attained: the *how*, the *what*, and the *why* of objects. The *how* is the class of the object, describing its internal implementation. The *what* is the type of the object, describing its external behavior. The *why* is a new concept call Role, which represents the task of the object within the organized structure of objects.

The method developed at SI consists of five main parts. Each part represents a systems development phase as well as part of the total description of the application under development. Objects are at the center of attention at all time; each part provides some information about the application objects or their structures.

The *Role Model* part separates the problem domain into more or less overlapping *areas of concern.* Each such area is modeled as a structure of interacting objects. Each object is abstracted into a Role according to its purpose in the Role Model structure.

The *Object Specification* part integrates the individual Role Models by letting a single *composite object* play different Roles in different models.

The *Class Implementation* part provides programs for all required objects.

The *Structure Specification* provides a kind of grammar or *Meta Model* that describes the possible collaborations between objects, (i.e. how they can be configured). This process is like multi-dimensional dominoes, where any piece may be attached to any other piece if they both have free and compatible interfaces where they can be joined.

Finally, the *Object Instantiation* part creates objects and interconnects them according to the prescriptions given in the Meta Model, and as instances of classes programmed in the Class Implementation part.

Role Modeling

Modeling consists of two subparts: analysis for modeling subproblems, and synthesis for joining small models into larger ones. The task in object-oriented design is to describe patterns of interactions and to assign responsibility to individual objects in such a way that the total system of objects is as simple as possible. An object that is to play a certain Role in an object structure must understand certain messages (e.g., have certain behavior).

Analysis: Simple Role Modeling

There is a many-to-many correspondence between Role and Behavior. For example, consider a document modeled as a structure of objects. Suppose that the documents tree has a document object as its root, and a number of sub-objects as shown in Figure 5.

This example has nine different Roles, but several objects may be given identical behavior. There are only three kinds of object behaviors:

a general **TreeObject** that can play the Roles of *document, title page, section,* and *figure*; a **TextObject** that can play the Roles of *title, author, paragraph,* and *caption,* and a **PictureObject** that can play the Role of *picture.* Thus, considerable code reuse has been achieved by separating the concepts of Role and Behavior. Furthermore, programs can be written to implement a great variety of document structures: many different Role Models may be constructed from a toolkit containing a limited selection of three different object behaviors.

Each Role in a Role Model is given a name. Its responsibilities are described to determine which other Roles it needs to know about and what messages it sends to these collaborators. While during execution the origin of a certain message is irrelevant to an object, the right to send a certain message is a very important part of the privileges that are assigned to objects in design.

The diagrams used for Role Models are very simple. Computer-based tools support drawing Role Model diagrams, and more detailed information is always immediately available to the designer through a direct manipulation tool interface.

For example, the *document* Role Model describes what is meant by a *document* in object-oriented terms. Objects that serve the same purpose in the model, for example *section* objects, have been abstracted into Roles. Attributes to the symbols in the diagram give further information, such as a description of the responsibility of the objects for each Role, and details about messages that objects may send to collaborators.

Synthesis: Composite Role Modeling

Typical designs are usually too large to be comprehended as a whole. By subdividing into subareas of concern, and creating Role Models for each subarea, a problem can be decomposed. This reduces modeling to manageable proportions, but creates a new problem of integrating smaller models into a model of the entire system.

This problem can be simply solved in the few cases where the problem can be considered hierarchical. What appears as one object on the higher level is then represented by its own Role Model on a lower level.

However, problems are usually more complex. If one model is not just detailing the internals of a single object in another model, a *Role Synthesis* construction mechanism is needed to integrate models. For example, given a number of Role Models A,B,... with Roles A1, A2, A3,..., B1, B2, B3,..., a *Composite Role* AnBm.. can be created such that the newly created Role object may simultaneously play Role n from Model A, Role m from Model B, and so on.

Thus a many-to-many correspondence between Role and Object exists, because an object may play several different Roles, and a given Role may be played by different objects. For example, a *person* object may play the Role of a *materials provider* in a manufacturing Role Model, and the Role of a *buyer* in a materials purchasing Role model. The *person* object then acts as an integrator between the two Role models, using knowledge about the market in its *materials provider* Role, and knowledge about manufacturing in its *buyer* Role. Another example is illustrated in Figure 6.

Role Synthesis makes it possible to reuse Role Models. Consider the document example. The *document* Role could inherit properties of the *parent* Role in the Tree Model, the *titlePage, section* and *figure* Roles could inherit both *parent* and *child,* and the leaf nodes *title, author, paragraph, picture* and *caption* could inherit the *child* Role.

The advantages of Role Synthesis are threefold. First, a tree structure does not need to be reinvented every time one is needed. Second, if tree structures need additional properties

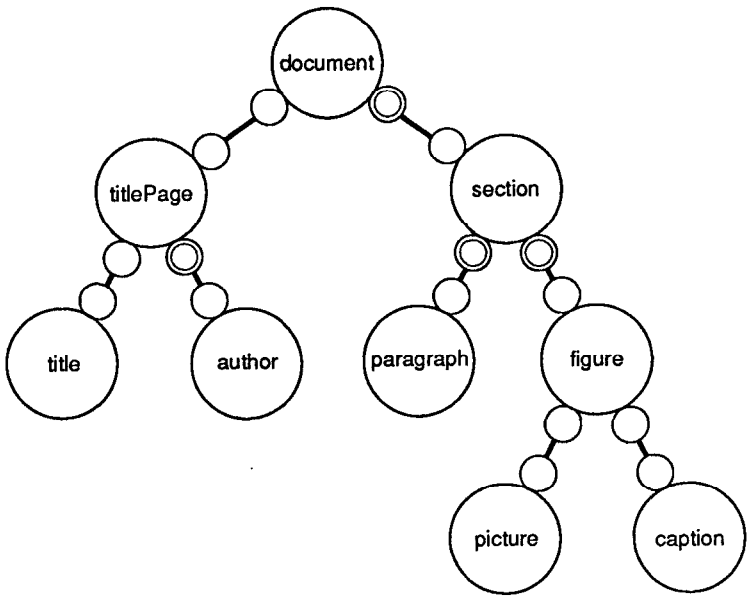


FIGURE 5. A Role Model diagram for the document example.
A Role is denoted by a large circle, collaborators are joined by lines, the small circles at the line ends denote the cardinality: A double circle denotes that for one of the near Role there are none, one or many of the far Role; a single circle denotes that for one near Role there is exactly one far Role. A symbol inside a small circle (not shown in the figure) denotes the messages that the near Role may send to the far Role, no symbol means that the near Role does not send any messages to the far Role. In the computer-based tool, any symbol may be selected to obtain a new window with further information.

then it can be added in only one place, namely the Role Model of a primitive tree. Third, classes for the *parent* and *child* Roles could be implemented; the document objects and all other used of tree structures could then be programmed as subclasses of these. This provides a mechanism for describing a class library on the abstraction level of modeling and design.

Object Specification

Role Modeling studies objects and their interactions. To create an Object Specification, the focus changes from the overall structure of objects to a single object and its immediate collaborators. Again, synthesis may be used to create specifications for objects that may play multiple Roles.

For example, consider the *paragraph* Role in a document. There could be two very different kinds of objects that could fill this Role. A PlainText could contain text local to the current document. A DatabaseText could represent the latest version of some text record existing in a database. Every time the document was printed or inspected, the latest version of this record would be inserted. And, if a user were allowed to edit such a text, the database should immediately be updated.

A DatabaseText object would need to play some Role such as *database record* in a Role Model describing a database system. It would, of course, also have to play the Role of *paragraph* in the *document* Role Model. These two Role Models could be combined into one, but this would create unnecessary complexity.

Instead, a better solution is to specify DatabaseText as an object that can both play the Role of *paragraph* in the *document* Role Model and *database record* in a *database* Role Model. This specification defines the object by describing all the Roles it must be able to play and all its interaction with its collaborators. This is illustrated in Figure 7, where the object being specified is shown in the center of the diagram with its collaborators around it.

Interdependencies between Role Models must also be considered. In the DataBaseText example, messages in the document domain having to do with getting and putting text will presumably have to perform some database operations. Conversely, if the database content is changed, some action should be taken in the document domain to reflect the new values. Such interdependencies are recorded in relevant message descriptions.

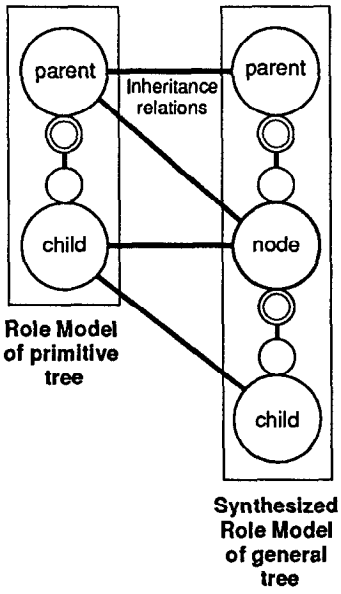


FIGURE 6. The general node object of a tree structure can be synthesized as a Role that can play both the *parent* Role and the *child* Role.
In the left part of this example, we create a Role Model modeling a primitive, two-level tree: One *parent* Role collaborates with any number of *child* Roles. We can now synthesize a Role Model for a general tree by letting all intermediate tree nodes inherit behavior both from the *parent* Role and the *child* Role. Intermediate Roles, *nodes*, may therefore play parent Roles vis-a-vis their children, and *child* Roles vis-a-vis their *parent* as shown in the right part of the figure.

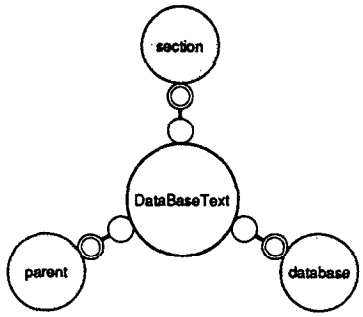


FIGURE 7. An Object Specification. An Object Specification describes all the Roles an object must play and thus the combined set of interactions with all collaborators. The object is shown in the center, its collaborators are shown as shaded circles around it. In this case, the Role is a *paragraph* of a *section*, the *child* of a *parent*, and a *record* of a *database*.

Class Implementation

The word *class,* in OORASS, is used in a very restricted sense: a class is a program that implements a certain object specification. The class is the only place where the internal structure of an object is seen. Just as a class can implement objects that play several roles, many different classes can implement objects that play a particular role (i.e., there is a many-to-one relationship between classes and roles). The object being specified in an Object Specification is often synthesized as a composite of several Roles. This inheritance structure gives important hints as to a possible class hierarchy in a program. Specifically, reusable Role Models should be implemented as reusable class libraries.

Structure Specification

The Object Specification part defines the external properties of an object in sufficient detail to decide whether objects should be connected. Any object that satisfies the assumptions an object makes about a collaborator may be connected to that object and play the Role of its collaborator.

Given a reasonably rich and generic set of Object Specifications with at least one class implemented for each, clearly a variety of correct object structures can be built. Only some will be meaningful in the user domain. For example, in our document, the programs will tolerate *title pages* that come between two *sections* in

the middle of the document. However, this is not typically what the user expects.

Therefore, the Meta Model describes the subset of workable object combinations that have meaningful structures in the user domain. This description is used during Object Instantiation to control the generation of an actual object structure [36]. The Meta Model can contain other information, such as number restrictions, parameters to initialize the attributes of an instance to adapt it to play a certain Role, access restrictions, default display and formatting information, in addition to pure structural information.

System Instantiation

The System Instantiation part creates an actual object structure by matching classes defined in the Object Implementation with the prescriptions of the Meta Model. This is typically a dynamic process where new objects are being created and unused ones are garbage collected throughout the lifetime of the application.

An object may collaborate with any other object that has the desired behavior regardless of implementation. For maximum flexibility, the binding of class to collaborator is postponed until the moment that new objects are actually created. The program needing the new object knows the name of its Class Specification, a run-time mechanism matches this name to its preferred implementation and creates an instance of the corresponding class. This is similar to the LaLonde's use of exemplars [26].

Further Work

The OORASS method has been evolving over several years. Successful programs have been developed exploiting the class Implementation, Structure Specification and Object Instantiation tools. A number of different systems have been generated from an identical program base by defining different Meta Models. New capabilities have been added by just programming the new classes and including them in a Meta Model.

Currently, a group consisting of people from SI, Taskon A/S and the University of Oslo is developing a method for systems analysis and model description, employing message scenarios and formal protocol definitions as an extension to the Role Models. They also are integrating their CASE tools to provide a seamless model of all information. This will also include an extension of their literate programming facility, so that formal and informal information can be intermixed [42]. They hope that their method will provide a high-level model description of reusable classes that could become the technical foundation for a marketplace of reusable class libraries.

Frameworks—Reusable Designs

One of the main advantages of object-oriented programming is that it supports software reuse. It is easy to see how object-oriented programming makes program components more reusable, but in the long run the reuse of design is probably more important than the reuse of code. Although abstract classes provide a way to express the design of a class, classes are too fine-grained. A framework is a collection of abstract and concrete classes and the interfaces between them, and is the design for a subsystem. Abstract classes are fairly well understood, but much less has been written about frameworks, and there is much less of a consensus on them.

The first widely used framework was Model/View/Controller, the Smalltalk-80 user interface framework [25]. It showed that object-oriented programming was ideally suited for implementing graphical user interfaces. MacApp is a later user interface framework designed specifically for implementing Macintosh applications [46]. It is actually a framework for all aspects of Macintosh applications, such as printing and storing documents on the disk. Recently there have been a number of user interface frameworks from universities, such as the Andrew Toolkit from Carnegie Mellon University [38], InterViews by Mark Linton at Stanford [32, 54] and ET++ from the University of Zurich [55, 56]. Each of these frameworks improves the state of the art in user interface framework design in some way, building on the successes and lessons of earlier systems.

Frameworks are not limited to user interfaces, but can be applied to any area of software design. They are one of the main reasons that object-oriented programming has such as a good reputation for promoting reuse. However, frameworks are different from simple class libraries, and require more work to design.

Frameworks

The idea and terminology of frameworks were developed at Xerox PARC by the Smalltalk group. Peter Deutsch describes frameworks in [11] (and less thoroughly in [10]). He emphasizes that the most important aspect of a framework that is reused is the interface or specification of the components. Although frameworks reuse implementation as well, reuse of interface design and functional factoring is more important because they constitute the key intellectual content of software and are far more difficult to create or re-create than code. This is the key insight behind frameworks.

Just as an abstract class is the design of a concrete class, a framework is the design of a subsystem. It consists of a number of abstract and concrete classes. (Deutsch uses the term "single class frameworks" and "multiclass frameworks", but we instead say "abstract class" and "framework"). Part of the definition of each abstract class is its responsibilities. In addition, a framework consists of the collaborations between the objects in its abstract classes.

Like a subsystem, a framework is a mixture of abstract and concrete classes. It differs from a subsystem by being designed to be refined. It can be refined by changing the configuration of its components or by creating new kinds of components (i.e.,

new subclasses of existing classes). A mature framework will have a large class library of concrete subclasses of each abstract class, so that most of the time an application can be "plugged together" from existing components. Even when new subclasses are needed, they are easy to produce because the abstract superclasses provide their design and much of their code.

For example, the user interface subsystem of a Smalltalk-80 application is almost always produced with the Model/View/Controller user interface framework. It will be built by connecting views and controllers together and parameterizing them with menus, messages to send on particular events, etc. Even when an application requires one or two new user interface classes, most of the classes in the user interface will come from the standard class library.

Brad Cox has likened reuse in an object-oriented system to integrated circuits and has advocated the use "software ICs," which are black-box components that can be used in a variety of contexts [7]. However, designing a framework is more like designing a family of chips or a logic family. The individual components are less important than the standard interfaces they share, and designing the interfaces is harder than designing individual components.

Most frameworks will be domain dependent. Although most of the publicized frameworks focus on user interfaces, frameworks can be used for much more than just user interfaces. User interface frameworks are popular in part because they are relatively domain-independent, are useful to most programmers, and correspond to a traditional computer science area of specialization. However, most subsystems will be application-dependent, so the frameworks that generate them will be too. Good examples are frameworks for VLSI routing algorithms [16], or for controlling real-time psychophysiology experiments [14]. Thus, most frameworks will be of interest only to application programmers working in a particular area.

There are several projects at the University of Illinois at Urbana-Champaign to design frameworks. The TS optimizing compiler for Smalltalk has a framework for code generation and optimization [23]. Code optimizations are never completely machine-independent; a framework for code optimization allows the compiler designer to easily build a customized optimization phase. The FOIBLE framework for visual programming environments provides a customizable graphics editor to which a visual language designer can add an interpreter, resulting in a visual programming language [21].

Choices is an object-oriented operating system written in C++ at the University of Illinois under the direction of Roy Campbell. It is more than just an operating system; it is an operating system framework. It consists of interlocking frameworks for file systems [33], virtual memory [44], communication [61], and process scheduling [43]. The file system framework shown in Figure 8, which was developed primarily by Peter Madany, has been used to implement a number of different file systems, including BSD and System V, MSDOS, a log-based file system an object store, and archive files.

The Inner File System Framework

One of the central classes of the file system framework in Choices is MemoryObject. A MemoryObject is a sequence of identically sized blocks. It is responsible for reading and writing its contents one or more blocks at a time. It also is responsible for maintaining the number of blocks that it contains. Thus, the key operations provided by MemoryObjects are *read, write,* and *size.* Many parts of a file system are MemoryObjects, such as files and disks. MemoryObjects are the part of the file system framework that is most used by the rest of the operating system. In particular, the virtual memory system also uses MemoryObjects, so they act as an interface between the file system and the virtual memory

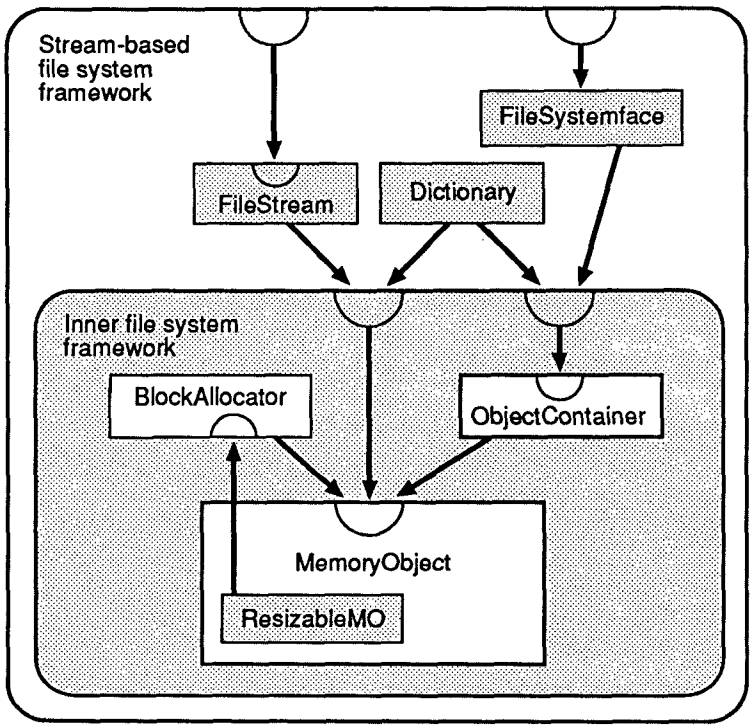


FIGURE 8. The Choices file system framework.

system. Files are stored on disks, but both files and disks are MemoryObjects.

ObjectContainers keep track of partitioning a large MemoryObject into a set of smaller MemoryObjects, (i.e., a disk into a set of files). For example, the Unix i-node table is an ObjectContainer. ObjectContainers can create new MemoryObjects, delete old ones, and can return the i-th MemoryObject that it stores. Thus, its operations are *create, delete,* and *open.*

A BlockAllocator manages the free blocks of a MemoryObject that is partitioned by an ObjectContainer. Of course, some MemoryObjects have fixed partitions and so have no free blocks, but most file systems allow files to be created and deleted dynamically. Files use allocators to acquire, and release blocks of the MemoryObject on which they are stored. The only operations defined by BlockAllocator are *allocate* and *free.*

MemoryObject, ObjectContainers, and BlockAllocators provide the foundation of any file system. Some of the components of a file system will be reused unchanged from the class library. For example, a disk can be partitioned by an ObjectContainer into a fixed number of fixed-sized file systems. Others are subclasses of standard abstract classes. For example, a Unix System V file system will consist of a subclass of ObjectContainer (SysVContainer) to implement the System V i-node table, a subclass of MemoryObject (SysVInode) to implement System V i-nodes, and a subclass of BlockAllocator to implement the System V free list.

Since a System V file system has a subclass for every abstract component of the framework, it might seem that the framework is not helping very much. However, not only is it very helpful for the designer to start with a high-level design that describes the components and their interfaces, but some of the concrete classes inherit a lot of code from their abstract superclasses. In particular, SVIDContainer inherits operations from ObjectContainer to manage a table of open MemoryObjects and from its superclass UnixContainer to read and write disk inodes, while SVIDInode inherits most of its operations from UnixInode.

Outer File System Framework

MemoryObject, ObjectContainer, and BlockAllocator are only the core of the file system framework. There is another layer that represents an application program's view of the file system. An ObjectDictionary (e.g., directory) converts a logical file name into the index of the file in an ObjectContainer. Its operations are *open, create,* and *delete.* A Directory differs from an ObjectContainer because a MemoryObject can be in only one ObjectContainer, but can be in many directories. Directories are usually associated with a single ObjectContainer, and are usually stored on the same MemoryObject that the ObjectContainer partitions.

There are currently two separate but compatible user views of files. FileStream provides a Unix-like interface to a file, with a current position and a *seek* operation, in addition to a *read* and a *write.* There is also a PersistentObject class that supports the transparent storage and retrieval of objects from the disk. Like FileStreams, PersistentObjects are based on MemoryObjects. They differ from FileStreams in that they can refer directly to other PersistentObjects.

Other parts of the file system include MountTable, SymbolicLink, FileSystemInterface (which keeps track of a current directory), and an authentication system.

This layering of frameworks is common, and is similar to other ways that software designs are layered. Some of the classes in the file system framework are used in other parts of Choices. MemoryObject is used by the virtual memory system, while FileSystemInterface and the user views of files such as FileStream and PersistentObject are used by application programs. BlockAllocators and ObjectContainers are usually private. The user of a file system does not need to know about the private classes, but the designer of a new type of file system does.

The Choices file system has gone through many versions, and each version is more general and reusable than the previous ones. The core classes have been stable for some time, while the outer classes are newer and still changing. This is typical of reusable designs. Reusing the early versions points out design weaknesses that must then be corrected. A framework's designer can be confident of its reusability only after it has been successfully reused several times.

In the fall of 1987, before the file system framework had been designed, a dozen students in an operating system course built a System V compatible file system for Choices. In the fall of 1989 two teams: a one-person team, and a two-person team, each built a log-based file system for Choices using the framework. The System V file system specification was simpler and much better documented than the log-based file system, but the students using the framework were more successful than the earlier students who did not use it. This experience not only increased confidence in the reusability of the file system framework, it illustrates why frameworks are so important.

Research on Frameworks

Designing a framework is itself research. The designer must understand the possible design decisions and must organize them in a set of classes related by the client/server, whole/part, and subclass/superclass relationships. Thus, the designer is developing a theory of the problem domain and expressing it with an object-oriented design.

There are three main research areas related to frameworks. The first is designing frameworks: what are the characteristics of a good framework and how is one designed? The second is using frameworks: how does one configure a particular application based on a framework. The

third is describing frameworks: what notation is needed, other than that applicable to object-oriented design in general?

Designing Frameworks

Most people realize they need a framework when they notice similarities in existing applications. These do not have to be object-oriented systems; designers with extensive experience in an application domain often can tell which frameworks would be useful for their applications. Subsystems also may evolve into frameworks as they are reused. Designing a good framework is more than just extracting the abstract classes from a subsystem. A subsystem only has to work for one application, but a framework must work for many applications. Thus, a framework is a generalization of the subsystems that can be built from it.

Good frameworks are usually the result of many design iterations and a lot of hard work. Designing a framework is like developing a theory. The theory is tested by trying to reuse the framework. Unsuccessful experiments require a change in the theory. Lack of generality in a framework shows up when it is used to build applications, so its weaknesses cannot be found until after it is designed and reused. Thus, iteration seems necessary.

Since iteration is necessary, it should be performed as early in the design life cycle as possible. Many iterations can be done on paper before any code is entered in the computer. However, the proof of the adequacy of a design is whether it can be implemented well.

Changes made to a framework during its design tend to fall into certain patterns [22]. Responsibilities are moved from one class to another. Responsibilities (or even classes) are broken into smaller components, so that one part can be changed independently of another part. Sometimes separately designed classes are given a common superclass, which is usually followed by migrating functionality up into the new superclass and preceded by renaming operations on the different classes so that they will share more of their interface. These changes are an important part of the process of designing frameworks.

The Software Refactory project (William Opdyke and Ralph Johnson) at the University of Illinois is developing tools to manage changes that occur during design iteration [37]. Refactorings are the changes to frameworks that do not add functionality, but instead redistribute and reorganize it. Refactorings are time-consuming and error-prone when done by hand, but the Software Refactory project is designing tools to automate them. Thus, a programmer will just perform the "break class into components" operation and both the class and its clients will be modified. This would make iteration much easier and would let the designer think about changes to a design at a high level. Moreover, changes to a framework can be propagated to the applications that use it. The result is that changes to a framework will cost less and have a larger benefit.

Using Frameworks

Using a framework is typically comprised of two activities:

- defining any new classes that are needed, and
- configuring a set of objects by providing parameters to each object and connecting them.

Ideally, no new classes are needed. Frameworks are seldom ideal; most applications must define new classes within the framework. However, even when new classes are needed, most of the work of using a framework is "plugging" or configuring objects together.

Programs that configure a set of objects are very stylized. First, a set of objects is created and each object is initialized. Then, operations are performed on the objects to connect them. These programs are so similar to each other that it is natural to think that they can be written automatically. Several of the user interface frameworks have tools that will automatically write the code to configure a user interface. Glazier was developed at Tektronix to build subsystems based on Model/View/Controller [1]. The NeXT Interface Builder is a much more powerful tool that builds user interface subsystems from the NeXT user interface framework [53].

Although specialized tools for configuring applications for particular frameworks are valuable, what is really needed are tools that can configure applications for any framework. Scripting languages, which are compact notations for constructing applications from existing software components, are a proposed solution to the problem. An object-oriented scripting language can serve the same role for a framework that a shell or a job control language can serve in a conventional environment. Two object-oriented scripting languages have been developed at the University of Geneva, the Visual Scripting Tool [24] and TEMPO [8]. TEMPO is specialized for applications that deal with concurrent activities and temporal relationships between them. The Visual Scripting Tool is a visual programming language (i.e., it is based on pictures instead of text).

Describing Frameworks

Since most object-oriented programming languages provide no direct support for either abstract classes or subsystems, it is not surprising that there is no good notation for describing frameworks. Frameworks are more than just the classes that they contain, but include instructions for making subclasses and for configuring applications from the frameworks. The ITHACA project is an Esprit II project involving a number of European companies, research organizations, and universities to design and build an integrated application development and support environment based on the object-oriented programming approach [40]. One of the goals of the project is a formalization of frameworks.

The goal of the ITHACA project is to reduce the long-term costs of application development for standard

applications from selected application domains. Besides a kernel of object-oriented languages and compilers, it will have an application development environment consisting of a set of programming tools and an object-oriented software information base. Only the software information base is tailored to a particular application domain. Initially there are four target domains: public administration, office automation, financial applications, and chemistry.

A key idea in the ITHACA project is the *generic application,* which is similar to what we have called a framework. The software information base contains generic applications and other software components, as well. Users of the ITHACA environment will have two kinds of roles: that of the *application engineer* who tailors an ITHACA environment to an application domain, and that of the *application developer* who generates specific applications using the components in the information base created by an application engineer. The application engineer is concerned with all phases of software development from requirements analysis to coding and validation, but does so for generic applications rather than specific ones. The application developer starts with a generic application and configures the final application from the available software components to meet application-specific requirements.

The software information base is more than just a class library because it contains and organizes the application domain model, requirements, specifications, software components, and documentation. The primary mechanism for organizing the software information base is the *frame,* which collects and organizes all information pertaining to an application, whether generic or specific. Generic applications are also specified with frames. There is a hierarchy of frames, from generic to specific. An applications developer builds frames for specific applications by selecting a generic application frame and filling in the missing information, such as new requirements and the resulting design choices.

An important part of the expected life cycle is to reevaluate generic application frames to improve their reusability. This might require that new software components be added to the software information base. This corresponds closely to the way that frameworks tend to be designed. In fact, generic applications are quite similar to frameworks, with the primary differences being that the ITHACA project is trying to explicitly capture requirements and specifications, while these typically are described informally in the documentation of most frameworks. The iterative nature of object-oriented design is addressed in work on recognizing class hierarchies, leading to algorithms that automatically restructure a class hierarchy upon introduction of one or more classes [5].

Demeter

The goal of the Demeter project, led by Karl Lieberherr at Northeastern University, is to develop CASE tools and their theoretical foundations to improve the productivity of object-oriented designers and programmers [31]. One of the key ideas is the class dictionary, which is similar to a grammar. Class dictionaries describe the part-of and inheritance relationships between classes. They can be interpreted as class definitions, type definitions, or grammars. Different tools within the Demeter system use different interpretations of class dictionaries to automatically construct various kinds of programs, such as class definitions, customized print routines for each class, and parsers that can read object descriptions from a text file and convert them into objects.

A class dictionary defines the structure of a class; a class module adds interface definitions. Class dictionaries are language independent, but method definitions in class modules are written in particular language. Demeter automatically produces programs from class modules. It creates the class definitions, printers and parsers and adds the code in the class modules to them. Demeter currently produces either Flavors or C++ programs.

Class Dictionaries

Demeter classes fall into three categories: construction classes; repetition classes; and alternation classes. A class dictionary can be parametrized by other classes and can inherit from other classes. For example:

```
CLASS Basket HAS PARTS
  content: Sequence(Fruit)
  weight: Number
END CLASS Basket.
```

defines a construction class named Basket with components content and weight.

An example of a repetition class is class Sequence. Sequence has a class parameter; S. A Sequence contains zero or more parts of the specified class:

```
CLASS Sequence(S) IS LIST
  REPEAT {S}
END CLASS Sequence.
```

An example of an alternation class is class Fruit:

```
CLASS Fruit IS EITHER
  Apple OR Orange
  COMMON PARTS
  weight: Number
  HAS INTERFACE
    VIRTUAL cost() RETURNS
    Number
END CLASS Fruit.
```

where Apple and Orange are probably simple construction classes. An alternation class is always an abstract class. Thus, there is no need to generate functions to construct objects for it. For example, in the definition of cost, the keyword VIRTUAL means that cost may be redefined in an Apple or Orange.

Software Evolution

Demeter takes advantage of its grammar-based foundation to provide tools that help plan the evolution of software [29]. Class modules are usually implemented according to a growth plan that is determined by a class dictionary. A growth plan for a class dictionary D is a sequence of increasingly larger subclass dic-

tionaries, starting with a smallest subclass dictionary of D.

Each implementation phase completes the method definitions for some subset of the classes specified in the class module and provides a test suite for debugging the code. The order of the phases and the classes involved in a phase are chosen in a way that each phase is an extension of the previous one. Each phase can successfully execute test cases for earlier phases. Method definitions are added incrementally until the application is completely implemented. Thus, a growth plan provides for many small steps in the development process, and each step can run all the tests of the first.

Another way to think of this approach is that each phase corresponds to an increasingly detailed prototype. At each phase, the prototype works on at least some of the objects. Later phases should work on all the objects that earlier phases worked on. Lieberherr has shown that the growth plan problem is NP-hard, but has also developed useful heuristics for it that are satisfactory in practice [28].

Constructing Classes Automatically

One of the advantages of the similarity of grammars and class dictionaries is that Demeter can infer class dictionaries from object descriptions [27]. Currently Demeter can abstract recursive class dictionaries from object descriptions. This problem is in general NP-hard, but for the special case of single-inheritance class dictionaries Lieberherr's group has developed an efficient algorithm. Objects must be hierarchical and cannot have cycles. For example, a particular basket of fruit might be:

<Basket> content: (<Apple> weight: 12, <Orange> weight: 4) weight: 16.

The task of constructing class dictionaries then becomes finding a grammar that accepts the object descriptions. The class dictionary must accept only object descriptions that are similar to the input set (i.e., the grammar that accepts every object description is not useful). Moreover, the class dictionary should be as small as possible, which prevents it from simply listing the original object descriptions.

The main motivation for constructing classes from examples is that specific examples are easier to invent than general classes. In the same way, parametrized classes are harder to think about than classes without class parameters. The Demeter system can automatically build parametrized classes from non-parametrized ones. Often the need for parameterized classes is not obvious until after the classes have been written, so tools to automatically parametrize classes are valuable for both design and maintenance.

The Law of Demeter

A common problem in object-oriented design is collaboration graphs that are too complex, that is, too strong of a coupling between classes. The Law of Demeter is a rule of good programming style that simplifies collaboration graphs and minimizes coupling between classes [30, 31]. Stated briefly, the Law of Demeter says that one should not retrieve a part of an object and then perform an operation on that part, but should instead perform the operation on the original object, which can implement the operation by delegating it to the part. The result of following the Law of Demeter is that a method depends only on the interfaces of its arguments and its instance variables, but it does not depend on their structure.

The Law of Demeter increases information hiding. Ideally, a class hides its implementation, but it is common for programmers to define an interface that lets clients of a class depend on its implementation details. As an extreme example, the only methods defined by a class might be accessing methods that simply read or write the instance variables. A more common situation is where a client can get a pointer to an array of components or to an internal hash table. Changing the representation of the class will then require changing all its clients. The Law of Demeter will not completely eliminate this, because it does not prevent a method from retrieving a component and then using it as an argument to another method, it just prevents the component from being the direct receiver of a message.

The Law of Demeter can be enforced; the Demeter system contains tools that will check whether a design follows the Law of Demeter. It reduces unnecessary object coupling and helps new programmers learn good programming style. This does not reduce the power of object-oriented programming, because any program can be transformed into a program that follows the Law of Demeter.

The Law of Demeter minimizes the coupling between classes and makes it easier to change a class interface. It increases information hiding by ensuring that one class cannot depend on the implementation of another. The Law of Demeter localizes type information. Thus, programs are easier to understand because each method depends on only a few interfaces. There are some disadvantages to the Law of Demeter [45], but it is an important contribution to object-oriented design rules.

The Demeter project is trying to find other rules of good design. Object interactions imply dependencies between the classes of the objects. These dependencies affect the reusability of classes and the cost of future development and maintenance of the software. Thus, rules like the Law of Demeter can have a big impact on the cost of developing software.

Conclusion

One indication that the work on standardization of terminology by the group at Hewlett-Packard is needed is the differences in terminology seen here. However, the fact that different groups are forced to invent terminology for the same concepts indicates that the concepts are important.

One set of similar concepts is interface description, contracts, and

role. These phrases all describe looking at an object by its specification. This is important because (as the Hewlett-Packard definition clearly states) classes are really implementation, and designers emphasize the way objects behave instead of how they are constructed. Thus, object-oriented designers need ways of talking about specification. "Contract" differs from the other two phrases because both the client and the server are included in the contract, while "interface description" and "role" emphasize the view of the servers. This difference might not be very important, since clients will refer to servers by their specification and not their class.

The Hewlett-Packard ensembles seem to be the same as the Tektronix subsystems. These phrases describe groups of objects that are designed to work together. This is important because it shows that classes alone do not provide enough structure for large systems. Therefore, additional structure is needed to organize how objects work together. None of the existing object-oriented programming languages or programming environments provide very good support for describing how groups of objects work together.

Scripting languages have the same purpose as the Structure Specifications and Object Instantiation of OORASS, and may indeed be the same. They emphasize the fact that an important part of creating an object-oriented application is connecting objects to each other and providing them with parameters. This will be an important area of research in the future, since the emergence of frameworks will mean that a larger fraction of programming will be configuring existing components.

Each design method described here differs from the others significantly. The Tektronix process has two main phases, exploration and a detailed design phase. The exploration phase uses modeling to find classes, responsibilities for each class, and collaborations between objects of different classes. The exploration phase is very iterative; finding a new responsibility might lead to new collaborations or classes. The detailed design phase determines inheritance, subsystems, and contracts, and focuses on building a design that will be as reusable as possible.

OORASS looks at a much larger part of the life cycle. It has five stages, and the first two, finding roles and object specification, cover the same part of the life cycle as the Tektronix method. It does not seem to provide as good a mechanism for automatically finding faults in the design as the Tektronix design process. However, the Tektronix design method could probably be adapted for use as the first two steps in the SI design process. On the other hand, OORASS not only includes class implementation (the third stage), but the structure specification stage and the object instantiation stage describe how applications are configured from preexisting parts.

The Demeter design process is to construct class dictionaries, convert them into class modules by adding interface specifications, and then to build and follow a growth plan to implement the classes. This differs from the other two design processes in emphasizing the structure of classes. Also, inheritance is emphasized from the beginning. Although all three design processes emphasize looking at examples, the Demeter system provides tools to automatically construct concrete classes from examples, and abstract classes from concrete classes. Thus, it is even more example-driven than the other two.

Although each of these methods has its own unique characteristics, they are more complementary than they are competing. Differences in vocabulary hide their similarities. As object-oriented design methods mature, they will borrow ideas from one another. The design methods of the future will integrate and expand on these ideas to support larger-scale design and composition at all levels and help object-oriented programming live up to its potential to make software more reusable and hence less expensive and more reliable. ■

References

1. Alexander, J.H. Paneless panes for Smalltalk windows. In *Proceedings of OOPSLA '87. SIGPLAN Not. (ACM) 22,* 12 (Oct. 1987), 287-294.
2. Apollo Computer. Network Computing System, Tech. Rep. I-27, 1987.
3. Beck, K. and Cunningham, H. A laboratory for teaching object-oriented thinking. In *Proceedings of OOPSLA '89. SIGPLAN Not. (ACM) 24,* 10 (New Orleans, Louisiana, October 1989), 1-6.
4. Bobrow, D.G., DeMichel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., Moon, D.A. Common Lisp object system specification X3J13. In *SIGPLAN Not.* (ACM) *23,* 9, (1988).
5. Casais, E. Reorganizing an object system. In *Object-Oriented Development,* D. Tsichritzis, Ed. Centre Universitaire d'Informatique, Universite de Geneve, 1989. pp. 161-189.
6. Coad, P. and Yourdon, E. *Object-Oriented Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1990.
7. Cox, B. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.
8. Dami, L., Fiume, E., Nierstrasz, O., and Tsichritzis, D. Temporal scripts for objects, *Active Object Environments,* D.C. Tsichritzis, Ed. Centre Universitaire d'Informatique, Universite de Geneve, June 1988, pp. 144-161.
9. De Champeaux, D. and Olthoff, W. Towards an object-oriented analysis technique. In *Proceedings of the Pacific Northwest Software Quality Conference* (September 1989) pp. 323-338.
10. Deutsch, L. P. Levels of reuse in the Smalltalk-80 programming system. In *Tutorial: Software Reusability,* P. Freeman, Ed. IEEE Computer Society Press, Washington, D.C., 1987.
11. Deutsch, L. P. *Design reuse and frameworks in the Smalltalk-80 system.* In *Software Reusability, Vol. II,* T.J. Biggerstaff and A. J. Perlis, Eds. ACM Press, pp. 57-71, 1989.
12. Ewing, J.J. An object-oriented operating system interface. In *Proceedings of OOPSLA '86* Conference Proceedings. *SIGPLAN* Not. (ACM) *21,* 11 (Portland, Oregon, November 1986), pp. 46-56.
13. Fishman, D.H. et al. Iris: An object-oriented data base system. *ACM Transactions on Office Information Systems, 5*-1 (1987), 48-69.
14. Foote, B. Designing to facilitate change with object-oriented frameworks. Master's thesis. University of Illinois at Urbana-Champaign, 1988.
15. Goldberg, A. and Robson, D. *Small-*

talk-80: The Language and its Implementation. Addison-Wesley, 1983.

16. Gossain, S. and Anderson, D.B. Designing a class hierarchy for domain representation and reusability. In *Proceedings of Tools '89.* (Paris, France, November 1989) pp. 201-210.
17. Halbert, D. and O'Brien, P. Using Types and Inheritance in Object-Oriented Languages. *IEEE Software* (Sept. 1987) 71-79.
18. Hewlett-Packard. *HP NewWave Reference Guide.* August 1989.
19. Hewlett-Packard. HP builds framework. Electronic Engineering Times. June 10, 1989. 73-74.
20. ISO. Information, retrieval, and transfer management for OSI. draft proposal. Part I: Management and Information Model. ISO/IEC JTC1/SC21 N, May 1989.
21. Jindrich, W.A. *FOIBLE:* A framework for visual programming languages. Master's thesis, Univ. of Illinois at Urbana-Champaign, 1990.
22. Johnson, E. and Foote, B. Designing reusable classes. *J. of Object-Oriented Program. 1,* 2 (June/July 1988), 22-35.
23. Johnson, R.E., Graver, J.O. and Zurawski, L.W. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88, SIGPLAN Not. 23,* 11 (San Diego, Ca., September 1988) 18-26.
24. Kappel, G., Vitek, J., Nierstrasz, O., Gibbs, S., Junod, B., Stadelmann, M., Tsichritzis, D. An object-based visual scripting environment. In *Object Oriented Development,* Tsichritizis, Ed. Centre Universitaire d'Informatique, Universite de Geneve, 1989. pp. 123-142.
25. Krasner, G.E. and Pope, S.T. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. of Object-Oriented Program. 1,* 3 (August/Sept. 1988), 26-49.
26. LaLonde, W. Designing families of data types using exemplars. *ACM Transactions on Programming Languages and Systems, 11,* 2 (April 1989), 212-248.
27. Lieberherr, K.J., Berstein, P. and Silva-Lepe, I. From objects to classes: Algorithms for object-oriented design. Tech. Rep. Demeter-3, Northeastern University, January 1990.
28. Lieberherr, K.J. and Holland, I. Assuring good style for object-oriented programs. *IEEE Software,* (September 1989), 38-48.
29. Lieberherr, K.J. and Holland, I. Tools for preventive software maintenance. In *Conference on Software Maintenance.* (October 16-19, 1989), IEEE Press, Miami Beach, Florida, pp. 2-13.
30. Lieberherr, K.J., Holland, I., and Riel, A.J. Object-oriented programming: an objective sense of style. In *Proceedings of OOPSLA '88 Conference. SIGPLAN Not.* (ACM) *23,* 11, (San Diego, Ca., September 1988) 323-334.
31. Lieberherr, K.J. and Riel, A.J. Demeter: a {CASE} study of software growth through parameterized classes. *J. of Object-Oriented Program. 1,* 3 (August/September 1988), 8-22.
32. Linton, M.A., Vlissides, J.M. and Calder, P.R. Composing user interfaces with InterViews. *Computer 22,* 2 (Feb. 1989), 8-22.
33. Madany, P.W., Campbell, R.H., Russo, V.F. and Leyens, D.E. A Class Hierarchy for Building Stream-Oriented File Systems. In *Proceedings of the 1989 European Conference on Object-Oriented Programming.* (July 1989, Nottingham, UK) S. Cook, Ed., Cambridge University Press. 311-328.
34. Meyer, B. *Object-Oriented Software Construction.* Prentice-Hall, 1988.
35. Miller, M.S., Cunningham, H., Lee, C., Vegdahl, S.R., The Application Accelerator Illustration System. In *OOPSLA '86 Conference Proceedings SIGPLAN Not. (ACM) 2,* 11 (Portland, Oregon, November 1986), 294-302.
36. Nordhagen, E., Generic Object Oriented Systems. In *Proceedings of Tools '89.* (Paris, France, November 1989) pp. 131-140.
37. Opdyke, W. and Johnson, R. Refactoring: An aid in designing application frameworks. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications,* September 1989.
38. Palay, A.J., Hansen, W.J., Kazar, M.L., Sherman, M., Wadlow, M.G., Neuendorffer, T.P., Stern, Z., Bader, M. and Peter, T. *The Andrew Toolkit—An Overview,* USENIX Association Winter Conference, Dallas, 1988.
39. Paseman, W. The Atherton Software Backplane: An Architecture for Tool Integration. *Unix Rev.* (April 1989).
40. Profrock, A.K. Tsichritzis, D., Muller, G., and Arder, M. ITHACA: An integrated toolkit for highly advanced computer applications. In *Object-Oriented Development.* Tsichritzis, D., Ed. Universite de Geneve, 1989, pp 321-344.
41. Reenskaug, T. and Nordhagen, E. The Description of Complex Object-Oriented Systems: Version 1. Senter for Industriforskning, Oslo, Norway, 1989.
42. Reenskaug, T. and Skaar, A.L. An Environment for Literate Smalltalk Programming. In *Proceedings of OOPSLA '89 SIGPLAN Not. (ACM) 24,* 10. (New Orleans, Louisiana) October 1989. 337-346.
43. Russo, V. and Campbell, R.H. Process Scheduling in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '88 SIGPLAN Not. (ACM) 23,* 11 (San Diego, California, Oct. 1988).
44. Russo, V. and Campbell, R.H. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89 SIGPLAN Not. 24,* 10, (New Orleans, Louisiana Sept. 1989) 267-278.
45. Sakkinen, M. Comments on the law of Demeter and C++. In *SIGPLAN Not. (ACM) 23,* 12 (December 1988), 38-44.
46. Schmucker, K. *Object Oriented Programming for the Macintosh,* Hayden, Hasbrouck Heights, New Jersey, 1986.
47. Shlaer, S. and Mellor, S. *Object-Oriented Systems Analysis.* Yourdon Press, 1988.
48. Snyder, A. Encapsulation and inheritance in object-oriented programming languages. *In Proceedings of OOPSLA '86 Conference. SIGPLAN Not. (ACM) 21,* 11 (Portland, Oregon, November 1986), 38-45.
49. Snyder, A. The essence of objects. Rep. STL-89-25. Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA.
50. Snyder, A. An abstract object model for object-oriented systems. STL-90-22, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA.
51. Snyder, A., Hill, W. and Olthoss, W. A glossary of common object-oriented terminology. Rep. STL-89-26, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA.
52. Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, 1986.
53. Thompson, T. The NeXT Step. *Byte 14,* 3 (March 1989), 265-271.
54. Vlissides, J.M. and Linton, M.A. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM User Interface Software and Technologies '89 Conference* (November 1989).
55. Weinand, A., Gamma, E. and Marty, R. ET++ –An object oriented application framework in C++. In *Proceedings of OOPOSLA '88 SIGPLAN Not. (ACM) 23,* 11 (San Diego, CA., September 1988), 46-57.
56. Weinand, A., Gamma, E., and Marty, R. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Program. 10,* 2 (1989), 63-87.

57. Wirfs-Brock, R.J. An Integrated Color Smalltalk-80 System. In *Proceedings of OOPSLA '88 SIGPLAN Not. (ACM) 23,* 11, (San Diego, CA., September 1988) 71-82.

58. Wirfs-Brock, A. and Wilkerson, B. Variables Limit Reusability. *J. Object-Oriented Program. 2,* 1 (May/June 1990), 34-40.

59. Wirfs-Brock, R. and Wilkerson, B. Object-Oriented Design: A Responsibility-Driven Approach. In *Proceedings of OOPSLA '89 Conference. SIGPLAN Not. (ACM) 24,* 10, (New Orleans, Louisiana, October 1989), 71-76.

60. Wirfs-Brock, R., Wilkerson, B., and Wiener, L. Designing Object-Oriented Software. Prentice-Hall, 1990.

61. Zweig, J. and Johnson, R. Conduits: A communication abstraction in C++. To be published in the USENIX C++ Conference, 1990.

Following is a list of contact names and addresses for the associated sections of this article.

Common Terminology
Contact: Alan Snyder
Hewlett-Packard Laboratories
P.O. Box 10490
Palo Alto, CA 94303-0971
snyder@hplabs.hp.com

Object-Oriented Analysis
Contact: Dennis de Champeaux
Hewlett-Packard Laboratories
P.O. Box 10490
Palo Alto, CA 94303-0971
champeaux@hplabs.hp.com

Research in Responsibility Driven Design
contact: Rebecca Wirfs-Brock
Tektronix, Inc.
P.O. Box 500, Mail Station 47-720
Beaverton, Oregon 97077
rebeccaw@tekig5.pen.tek.com

Object-Oriented Software Engineering
contact: Trygve Reenskaug
Senter for Industriforskning
P.O. Box 124 Blinden
0314 Oslo 3, Norway

Frameworks—Reusable Designs
contact: Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Ave.
Urbana, Illinois 61801-2987
johnson@p.cs.uiuc.edu

Demeter
contact: Karl Lieberherr
Northeastern University, College of Computer Science
Cullinane Hall, 360 Huntington Ave.,
Boston, MA 02115
lieber@corwin.CCS.northeastern.EDU

CR Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.10 [**Software Engineering**]: Design—methodologies, representation; D.3.3 [**Programming Languages**]: Language Constructs

General Terms: Documentation, Experimentation

Additional Key Words and Phrases: Object-oriented design research

About the Authors:

REBECCA J. WIRFS-BROCK is a principal software engineer at Tektronix, Inc., and coauthor of *Designing Object-Oriented Software* (Prentice-Hall, 1990). She has spent 15 years designing software and managing software products. She managed the development of Tektronix Color Smalltalk, and has developed and taught courses on object-oriented design. **Author's Present Address:** Tektronix, Inc., P.O. Box 500, Mail Station 47-720, Beaverton, OR 97077.

RALPH E. JOHNSON is an assistant professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He has extensive experience with object-oriented programming in both C++ and Smalltalk, having been involved with medium-sized applications, such as operating systems and an optimizing Smalltalk compiler. **Author's Present Address:** Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Ave., Urbana, Illinois 61801-2987.