Objects in Groups

Doug Lea
SUNY at Oswego / NY CASE Center
Oswego, NY, 13126, USA
(315)341-2688
`dl@g.oswego.edu`

December, 1993

Abstract

A *group* is defined as one or more objects bearing a common abstract relation, common external access policies, common connectivity, and common internal policies. Groups are similar to objects, but lack a single locus of control. They otherwise share features analogous to object-based classes, identities, constraints, and implementations. Groups may be used to organize, analyze, and implement large and distributed object-oriented systems, including those based on CORBA.

1 Introduction

An object-oriented application may consist of a sea of objects; perhaps thousands or millions of objects. Subsets of these objects often possess discernible structure that is not well-captured by common object- and class-based concepts and notations. This paper surveys a simple unifying construct, *groups* that meets many needs for describing and managing the statics and dynamics of networks of collaborating objects.

Briefly, a group is a special kind of *set*, consisting of one or more objects (*members*) bearing a common abstract relation, common external access policies, common connectivity, and common internal policies. In more detail:

- A group does not have a single locus of control.
- A group as a whole may provide services, defined via an interface, to clients.
- A group has an extent, a finite collection of members that may change over time.
- Each unique group may be ascribed a unique identity.
- All members of a group receive the same messages from group clients.
- Group members may communicate among one another in ways that are invisible to external clients.
- Eligibility for membership in a group may be restricted.
- An object may join (and leave) any group for which it is eligible.
- An object may be a member of many groups at the same time.

The above characterization semiformalizes the structure apparent in many everyday real-world groups and social organizations. Group structures are equally ubiquitous in software systems. The following examples illustrate some common usages:

Subscription Group. A group in which all members receive the same set of messages (or mailings, postings, etc), usually without any requirements to act in any particular way on those messages.

Work Group. A group (e.g., a department) in which each member performs one or more subtasks in support of one or more larger tasks.

Service Group. A group (e.g., of text formatters) in which any one member may handle any service request.

Resource Group. A group of functionally identical objects (e.g., printers) that may be reserved and released by clients.

Access Group. A group (e.g., a Unix^{tm} login group) in which each member has special priviliges with respect to other members of that group.

Replicate Group. A group in which each member responds in the same way to incoming messages, normally to obtain fault-tolerance: Even if a member fails, at least one answer may still be obtained.

Transaction Group. A transient work group (e.g., of certain bank account objects) in which each member obeys a protocol ensuring particular transaction semantics.

Property Group. A group of objects all possessing a possibly transient property (e.g., a group of visible windows).

Location Group. A group (e.g., a family) in which all members reside at a common location.

The concept of a group is by no means novel in OO (see, e.g., [17]) or other accounts of system design (see, e.g., [3]), and is in fact increasingly widespread. The present treatment differs from others mainly in attempting to integrate disparate accounts and properties, and providing a basis for more firmly entrenching basic group constructs within the OO paradigm, while avoiding premature formalism or commitment to details of how group-based constructs might be adopted within existing OO languages and notations.

The remainder of this paper proceeds as follows: Section 2 compares groups to other OO constructs. Section 3 discusses message passing in groups. Section 4 describes the definition and use of common interfaces for groups. Group membership and access control are discussed in Section 5. Finally, Section 6 illustrates the use of groups in CORBA systems.

2 Related Constructs

The concept of a group fills a gap in the analysis, design, and implementation of object systems. Groups do not themselves solve structuring problems, but do provide a way of talking about issues that escape the confines of purely class-based and object-based concepts and notations.

Modules

A group may be seen as an extension of the notion of a *module* (or *subsystem* or *namespace*). A module is a grouping of related objects and/or services under a common scope. However, modules are too limiting to be useful in defining the structure of many of the above example applications. Strict module containment results in static, "pyramidal" architectures, and cannot capture the fluidity of most OO designs. Groups provide the required extensions. The notion of a group entails the option that membership be dynamic. Objects may join and leave groups, and may be members of several groups simultaneously. Thus, group membership is not necessarily tied to static scoping. Also, group descriptions are multiply instantiable, enabling the existence of two or more discernibly different groups with the same features.

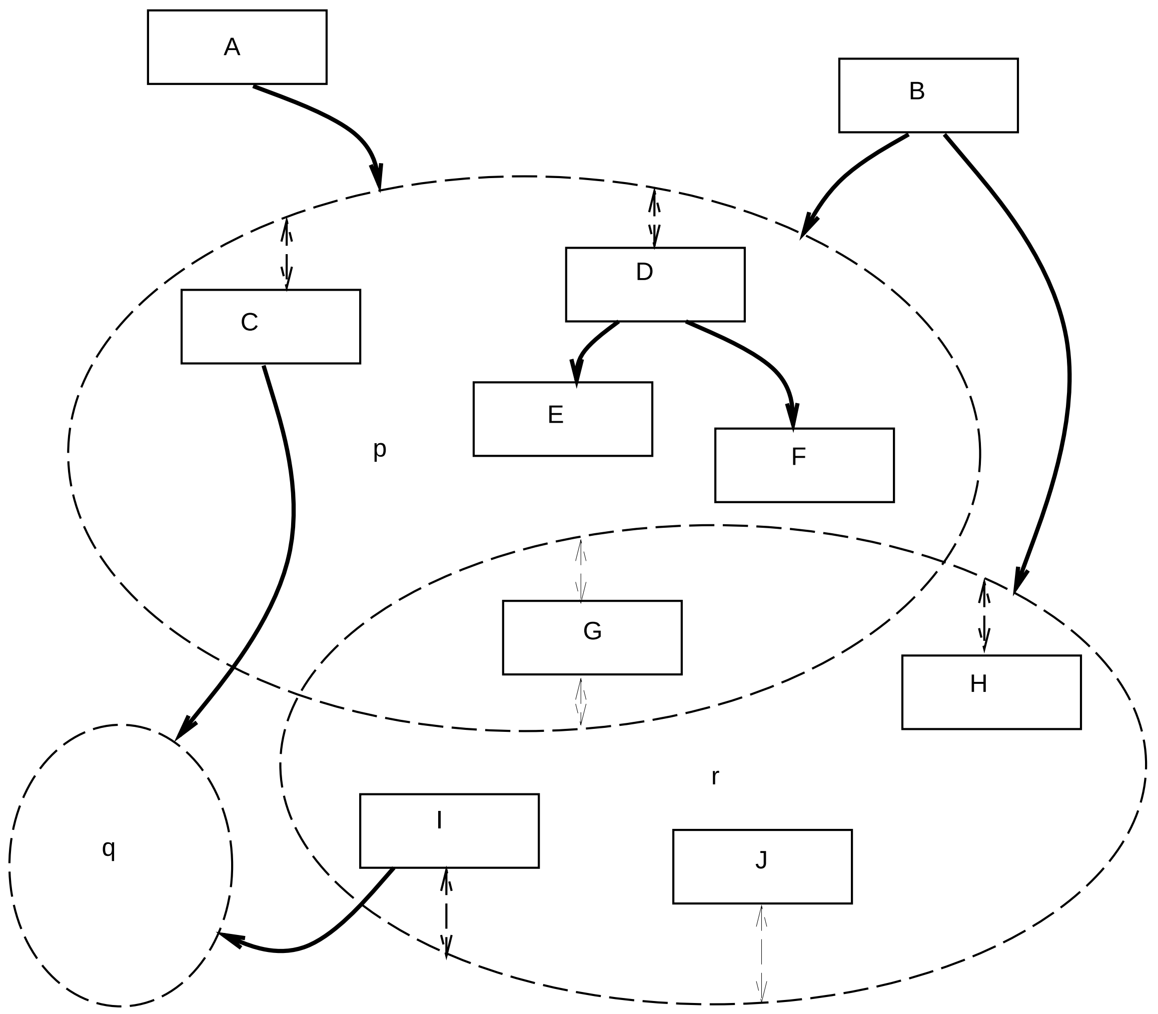

Figure 1: Some objects (A-J) in groups (p-r). Dashed lines represent group member connections; solid lines client connections.

Objects

While they share many features, groups are not themselves objects. Under any reasonable definition, an individual object possesses a unique locus of control. This applies even to composite (or ensemble[7]) objects. Although a non-primitive *host* object may communicate with other *acquaintance* objects in the course of processing messages, the host remains distinct from its acquaintances.

Groups, on the other hand, have no single locus of control or single message port. Control and functionality are instead distributed across member objects. Group-like constructs have found widest applicability in distributed systems, where groups of objects residing on different machines must somehow coordinate their efforts to achieve tasks without the benefit of centralized controllers. Evidence from over a decade of experience[3] in (non-OO) distributed systems, especially, suggests that groups will become central organizing constructs in the development of large OO systems.

Groups are by no means replacements for objects. When a unique locus of control is logically required, conceptually meaningful, and/or simpler to design, a group-based design is inappropriate. However, groups become central in designs favoring cooperation over control. Groups establish a layer of functionality without adding a layer of control.

If desired, a group may be "converted" into a composite object by superimposing a host. Any of the above examples of groups could be transformed into collection objects by superimposing a host *controllers* on the group, where the host exclusively manages the group in part by intercepting messages from foreign clients and directing them to the members. While sometimes desirable, such designs are useless, for example, in replicate groups employed for fault tolerance – a single host represents a single point of failure, defeating the purposes of replication.

There is in fact a continuum of coupling and control bordered by the members of the loosest kinds of groups at one extreme, and acquaintances of the most tightly coupled and controlled composites at the other. Broad categories of members include:

- Objects that can be ascribed membership in a particular group, but whose behavior in no way depends on this.
- Objects that modify their behavior if and when they are members of particular groups.
- Objects that behave differently depending on the nature and presence of other group participants.
- Objects playing particular roles in groups (e.g., by ignoring all but a special subset of messages).
- Objects delegated messages by possibly many controllers.
- Objects receiving messages *only* from single controllers.

Most differences in the design of groups versus objects are consequences of the fact that "internals" of groups are not contained within a single object. The state of a group is a function of the current members and relationships among their states. Group behavior is in part a function of its state. This is an extreme form of the dependencies seen in ordinary composite or delegated objects, where proper functioning of the composite relies on invariant relations and contracts[12] holding among acquaintances. It is more extreme for groups simply because there is no single controller present to maintain and ensure these invariants.

Sets

Groups are special kinds of sets. But unlike Agha's otherwise similar but purely intensionally-based set-like *ActorSpace*[1] construct, groups have both intensional and extensional components. Interfaces (Section 4) describe group capabilities and minimal features of members. However, two distinct groups may be constructed, each with the same interface, but with different members.

Collections

Groups serve as an alternative to various set-like collections entities that pervade OO systems. Groups are used in similar ways as common OO collections whose memberships are constrained by interface, but otherwise defined circumstantially within applications, although perhaps triggered via evaluation of inclusion predicates. In most cases, the notion of a group, rather than a collection object that controls it, provides a simpler basis for modeling and design.

Classes

A group is obviously different than a class. However, all instances of a class might be defined as members of a particular group. In this way those aspects of *metaclasses* responsible for tracking and managing instances of a given class may be reformulated via group constructs.

3 Channels

Two different groups must be considered as distinct even if both possess the same external features. Conceptually, groups, like objects, must be ascribed unique identities.

In the uninteresting case where group membership reflects a relation that has no consequences with respect to sending or receiving messages, the existence of group identities poses no further challenges – such groups serve only as labelled sets. But most OO systems employing groups require some means for clients to direct particular messages to particular sets of objects, and/or for the members of a group to interact. One way to do so within standard OO systems where messages must be targeted to particular recipients is to construct groups so that they appear as "objects" to clients ([4]). A more general route is to adopt *channel*-based, rather than object-ID (OID) based messaging.

A channel is an abstract transport medium that connects possibly many objects sending and receiving messages. Channels are distinct from the objects that are connected to them, and thus readily support messaging directed to groups and/or single objects without requiring that clients use different forms of message addressing in the two cases. In channel-based systems, clients may be unaware of whether messages are being consumed by groups or individuals (which may be construed as singleton groups). This in turn enables construction of transparent service replication, efficient service groups, and simpler interposition and instrumentation. Systems may for example split channels, interposing other objects between senders and receivers to deal with environmental constraints; e.g., for performing authentication or data conversion.

An object may be connected to several channels at the same time. This may be conceptualized and implemented by ascribing multiple mailboxes or listening ports per object. Alternatively, a single mailbox may be used for each object (as in Actor systems[2]), to which messages from all connected channels are deposited, possibly tagged by channel. Along another dimension, messages may be processed by objects using interrupt-driven, call-based, or polling techniques, and may establish priorities to messages on certain channels.

Channels and Identities

The use of channels does not otherwise impact most other practicalities in the design of OO systems. For example, channel names may be passed around in messages and used in the same way as standard OO references and pointers for directing messages to targets.

However, the use of channels versus OIDs represents a conceptual shift, in the direction of other theoretical and applied models of concurrent and distributed processing (e.g., [20, 28]). Channels help disambiguate identity from message targeting mechanisms. These may be collapsed in point-to-point systems, but not with groups. The fact that each object and group has an identity does not imply that they or others "know" that identity, but only that messages be sent along channels that are connected in the desired fashion. Many process notations do not even acknowledge identities of individual processes or objects. They instead indicate the attachment of entities and channels via static syntactic mechanisms.

Channel-based OO systems must still employ some kind of OIDs, even if they are not used for message targeting. A system must track identities (although perhaps implicitly) in order to support *binding* operations. OO systems employing channels require a primitive binding operation that causes an object to start sending and/or receiving messages along a given channel. A similar *unbinding* primitive disconnects the object from the channel.

A channel-based OO system may also support a dynamic *binding query* telling whether a given object is attached to a given channel (i.e., whether an object is a member of a group), and/or derived operations that check to see if two channels connect to the same object(s). Support for group membership queries enables common OO identity-based processing idioms, for example, those that dynamically test for the presence of aliasing and interference[14]. However, even with such support, such checks can at most test "top-level" identity, which fails to detect potential nonindependence of composite objects sharing access to other acquaintances or groups. This places additional burdens on design and implementation measures that otherwise preclude unwanted interference (see Section 5.2).

Implementing Channels

The separation of channels from identities facilitates the specification and implementation of "quality of service" issues surrounding message transport. Messages through channels connecting groups must be *multicast* to all members. Multicast should possess simple and predictable ordering semantics and be fault-tolerant. Minimally, multicast channels should have the FIFO property that messages are received in per-sender issued order. Causal and atomic broadcast protocols may be used to extend these guarantees to causal sequences across multiple senders[3, 21]. The strongest possible guarantee, full synchronicity, is normally undesirable since it limits parallelism and requires centralized message coordination.

In both sequential and distributed systems, underlying transport mechanisms are often point-to-point, in which case group channels and multicast support must be fabricated on top of other primitives. Multicast may be implemented through various *apply-to-all* constructs. This often amounts to the artificial imposition of collection objects that track and send messages to members. The *transparency* of these mechanics is a policy decision in any group-based programming system. Full transparency maintains and enforces the desired abstractions while permitting infinite implementation latitude.

Use of more visible implementation techniques permits simpler accommodation of groups in standard OO frameworks. Since OIDs and channels may be collapsed in point-to-point systems, a message to a channel may be implemented as an ordinary OID-based message to an object serving as a *proxy* for the channel. (In most distributed object systems, proxies are used even to implement point-to-point messages.) For example, a proxy could maintain a list of member addresses, and send point-to-point messages to each. Each process in a system must have local proxies. Proxies must themselves be managed, requiring additional infrastructure to track the existence and locations of members [27] and maintain consistency among proxies, again with a range of transparency options. Since several proxies could serve as channels to the same group, or even vice-versa, identity comparisons among channel proxies are meaningless at the application level.

4 Interfaces

As is the case for objects, the notion of groups invites the distinction of "external" and "internal" views of features and services. For objects, such matters may be addressed by separating external specifications of messages that may be received from foreign clients, versus internal constraints and computational descriptions of how messages are processed. The same principles apply to groups.

The use of channels provides a basis for separating either object or group external interfaces from internal matters. When communication occurs via channels, the client view of a group need not differ from that of an object. In both cases, messages must conform those supported by a channel. Thus, the capabilities of the channel itself may be defined via an interface that holds regardless of whether the channel is bound to an individual object or a group. However, the use of groups and channels does not strictly require the use of types or interfaces. A channel interface may be defined in an implicit or *post hoc* manner, as the intersection of the messages receivable by all potential members, or even via an arbitrary predicate constraining membership.

A channel interface description defines a *type*. Like object types, channel types provide a specification of the features (e.g., operation signatures) common to all instantiations of that type. Channel types are intrinsically bidirectional, describing the forms of messages received by members, and results sent back to clients. However, a channel and/or its type may also be split into descriptions of client-side versus member-side sending and receiving rights (cf., [28]).

A channel can represent the extension of a group; a channel type describes its visible features. Groups themselves do not require external interfaces distinct from their corresponding channels. However, channels connecting to groups may differ idiomatically from those connecting to single objects. For example, multiple replies are more common. When more than one member is expected to reply to a message, the interface may be described in terms of multiple results, streams, collections, or via the use of interposed filters that reduce messages to a single reply.

A *subinterface* may be defined by subtyping an interface, for example, to describe connections to individuals or groups offering additional services. Subinterfaces may be used in the definition of *subgroups*. However, the concepts of subinterfaces and subgroups do not bear a one-to-one relation. A subinterface is an extension of one or more base interfaces. Members of subgroups need not conform to particular subinterfaces. Subgroups may consist merely of circumstantially selected members of a parent group.

5 Membership

Interference (safety and/or liveness failures stemming from "unanticipated" interactions) can be a serious problem in OO systems. The best means for preventing and coping with interference is to control the inward reachability (fan-in), outward reachability (fan-out), and/or the bidirectional locality or closedness of a set of objects. Module constructs employ static scoping mechanisms to address such issues. Module membership is established by declaring an entity within module scope and external access is controlled via export constructs.

For groups, analogous dynamic constructs provide a basis for design policies, rules, and tools for limiting and controlling interference. For example, Wills [31] describes a collection of methods that restrict external communication by individual objects to others in particular *demesnes*. Demesnes, as well as the related concept of *islands* [13], represent formalisms of particular group-based constructs and policies aimed at simplifying the analysis and design of communications-closed sets of interacting objects by imposing restrictions on the groups with which individual objects may be members and/or clients. Unlike *ActorSpaces*, groups are not defined in a purely intensional manner, so enable/require the active control of membership. Not all objects conforming to a given interface necessarily belong to a particular group instantiation with that interface. Instead, members must join and leave groups explicitly.

There are two aspects of group construction, creating a group and enlisting members. Construction of a group simply amounts to the construction of a channel, which must be a primitive operation in any group-based system. (Section 5.3 describes the use of mulitple channels per group, a straightforward extension.) Destruction of a group deletes the channel. Destruction may be automated by garbage-collecting unreferenced channels.

The enlistment of new members into a group is very similar to the construction of new instances of a class (e.g., via `new` operators in C++ and Smalltalk), with the obvious differences that a new group member must already exist in order to join a group, and that an object may be a member of several groups at once and change its group membership over time.

Minimally, an object joins a group via the act of binding (as a receiver) to the channel transporting its messages, and leaves a group via unbinding. Group-wise versions of set operations may be built from these primitives; for example, an operation to union all members of one group into another via member-wise joins. One may also define and construct groups of groups (as opposed to groups of individuals).

In a fully channel-based system, binding is the *only* means of receiving messages of any kind, so group joins are routinely required upon object construction. These may include, for example, "metaclass groups" containing all objects of a given class and/or "location groups" (or *clusters*[7, 5]) containing objects constructed on a given machine. Also, each new object may construct and join a singleton group. Generally, objects of a particular class may be defined to belong to some predetermined groups throughout their lifetimes, belong to others only under certain predefined conditions, and/or join and leave others opportunistically.

5.1 Eligibility

Channel interfaces impose constraints on objects that may connect to them. All members of a group should be able to receive messages listed in the corresponding channel interface, although members playing special roles may ignore certain messages. In a strongly typed framework, eligibility to connect to a channel is statically checkable. Each member must be of a type (class) that *conforms*[25] to the interface. Interface types may be made arbitrarily specific as a means of limiting group membership, even to the extent of presenting *de facto* requirements that the group be *homogeneous*, as is common in fault-tolerant applications.

Membership may also be tied to contracts describing dynamic properties that escape static checks. For example, a certain group of windows may require that each member have its `visible` attribute set to `true`. Implementation of this constraint may require the use of standard OO mechanics, including per-member notification and update mechanisms that issue group joins and leaves upon changes in status, along with interception of attempted binds and unbinds using group managers that verify eligibility.

5.2 Management

While not all internal group matters can be ascribed to a single object, encapsulation goals require that matters common to all members be somehow localized. In the same way that object construction may be handled through the use of metaclasses or factories/generators[8, 7], it is profitable to centralize (with the help of infrastructure and/or policy support) membership control for a group by defining *group managers* (or metagroups[19]). A manager may be either a separate entity that is designated to handle binding, etc., or a member of the group that responds to special `join` and `leave` messages issued upon binding[30, 17]. In either case, managers remain distinct from controllers of the sort described in Section 2 since they do not intercept normal group messages. A group manager may perform other bookkeeping duties including:

- Screen potential members.
- Maintain membership lists.
- Periodically issue special probe messages to discover whether members are still alive.
- Notify other group members upon membership changes.
- Log group messages.

- Control resources shared among group members.

Group managers may also control external access. To become a client, an object must bind (as a sender) to a group channel and then start issuing messages through it. This provides the same opportunities for control and management as are available for group membership. Sometimes, client access may be restricted through static interface conformance checking. Most cases must be handled dynamically. Group managers may intercept binding requests to check for client properties, check against access control lists, engage in authorization protocols, and so on.

5.3 Roles

A *role* is defined by a set of features and services (or interface) employed in a particular context, but without committing to any specific object that must offer them (cf., [6]). Specific roles for helpers, subcomponents, or delegates of a composite object may be described in terms of constraints and contracts between the host and the acquaintances. However, in group-based designs, lack of host controllers causes the notion of a role to split into two aspects:

Public. An object may play a particular role in a system by virtue of being a member of a particular group.

Private. An object may play a particular role in a group by virtue of responding only to a certain subset of messages to the group, and/or by responding to them in particular ways.

In principle, this dichotomy may be transformed into a continuum via subinterfaces and subgroups. Conceptually, a system as a whole may be considered as a group, with subgroups serving as subsystems. These may in turn be subdivided into arbitrarily fine subgroups reflecting increasingly narrow roles. However, as with classes, pragmatics dictate that subgrouping "bottom-out" in conceptually meaningful categories, with remaining variation handled by defining private roles for individual objects.

Private Roles

The difference between the public and private aspects of individual *objects* may be expressed by encapsulating local processing within the boundaries of these objects (usually within the "private" parts of class descriptions). This trick doesn't work for groups. However, purely internal group communication can be segregated by establishing *inner* channels, that transport messages from and to members, not clients. Group members may be connected to two channels, the outer one for "public" messages, including those to and from clients, and the inner for "private" messages among group members. These effects may also be obtained using only one channel, tagging messages according to whether they were issued by clients versus members.

Inner channels may be employed to assist in synchronization and control of members. For example, locks may be issued to all members of a group performing an atomic transaction. Inner channels may also support protocols among members playing special within-group roles; for example:

Standby. Handling messages only if other members cannot.

Arbiter. Maintaining consistency; resolving lack of consensus among members.

Filter. Collating group results and submitting replies to clients.

Task Manager. Breaking up tasks into subtasks handled by other group members.

Of course, not all collaboration protocols can be handled in this way. Problems requiring supervised oversight lead to the superimposition of a single controller to receive incoming messages and manage communication among group members.

Public Roles

A given object may play many roles in a system, and change roles across time. While it is at best problematic to arrange that objects change their *class* membership to effect multiple, context-dependent, and/or time-varying roles, such capabilities are intrinsic to the notion of group membership.

The relationship between groups and channels can be exploited notationally in OO analysis, design, and/or programming to simplify the description of public role-specific behavior. Responses to messages defined in the channel interfaces of all groups to which an object could be a member may be segregated by channel type in its class description. When classes multiply inherit interfaces of multiple groups, and these include overlapping method sets, responses may be disambiguated on the basis of channel type. However, a different notational ambiguity/nondeterminacy remains possible if an object is a member of more than one group with the same channel type.

Membership and access control measures enable the selective export of particular group interfaces for usage by particular clients. Such channels represent *views* [26] of objects, permitting role-based, *subject-oriented* [11] design and programming methods. Group constructs also help tame extreme forms of object evolution encountered in such designs when objects need to acquire roles that were not even defined at the time they were constructed. Rather than permitting arbitrary class changes, flexible yet more tractable policies may be formed by restricting changes to the concatenation of new group interfaces to existing capabilities and enlistment in groups receiving messages on the corresponding channels.

The best-established means for supporting such maneuvers is to define *all* adaptable entities as special kinds of groups. A *fragmented object* [18, 9] is a group possessing an inner channel, but not an outer channel. Fragmented objects thus lack public group interfaces. Instead, some members (*providers*) possess possibly different externally accessible interfaces representing their public roles, and communicate via the inner channel with other group members in the course of serving client messages. A fragmented object thus appears as different objects to different clients and may grow to serve new roles by adding new providers with different external interfaces.

6 Groups in CORBA

As indicated in the course of this paper, basic definitions of groups (like those of classes and objects) provide great room for variation in expression, design methods, and implementation techniques. One can imagine syntactic and semantic accommodation of groups within common analysis and design notations (e.g., OMT) and OO programming language genres (e.g., CLOS, Smalltalk, Eiffel).

As an example, this section describes the use of groups in C++-based CORBA systems, by way of experiences in collaborating with Isis Distributed Systems, Inc., to develop a prototype toolkit adding basic group support to CORBA functionality. (This prototype is currently being transformed into an IDS product.)

The Object Management Group's CORBA (Common Object Request Broker Architecture) specification [22] defines an infrastructure for distributed object systems that is surprisingly amenable to group constructs. Although seemingly class-based, two key constructs (`interface`s and `ObjRef`s) in the CORBA Interface Definition Language (IDL) are defined in a way that apply equally well, if not better, to groups.

IDL `interface`s are very similar to C++ "abstract classes" [29]. They define sets of attributes, service procedures, and oneway (resultless) methods. These features must be implemented in a particular target language (often C++). However, IDL `interface`s are specifically not tied to single-object implementations, and contain no semantic requirements that interfere with their use as group interfaces. Thus, like channel types, they may serve either role. IDL contains no provisions for specifying contractual invariants among objects or other architectural information, so neither precludes nor supports group-based designs.

The basic IDL reference construct, the `ObjRef` is defined in a loose enough fashion to represent group channels rather than, or in addition to links to objects. While the semantics of `ObjRef`s do not yet appear to be completely defined ([24]), they bear much similarity to channels. They are typed by interface, and may be tied to distributed collections of services rather than identifiable objects. They also lack an intrinsic object identity test operator.

Despite these correspondences, CORBA defines only point-to-point message protocols, and includes no provisions for multicast. Group-like protocols are instead off-loaded to the CORBA Event Notification Service (ENS)[23] that is intended to be layerable on top of the ORB. The ENS itself defines several variants of "channels" as *interfaces*.

Our prototype tool, RDO/C++ (Reliable Distributed Objects in C++), re-bases basic CORBA functionality on top of the Isis^{tm} Toolkit[15]. Isis provides a library of implementation support (written in C) for channels, multicast, membership management, failure detection, fault-tolerance, and group monitoring, along with higher-level tools that assist in the development of group-based designs involving publish-subscribe, standby, spooling, and transaction protocols. Except for CORBA compatibility, RDO/C++ shares most features with other efforts that have integrated Isis (or its research follow-on, Horus[30]) into distributed OO systems tools[10, 17], including those of a companion effort, RDO/ST, that uses Isis to support distributed Smalltalk programming.

The overall structure of RDO/C++ is similar to that of most C++ distribution tools (e.g., [18]). The RDO/C++ IDL compiler converts IDL to C++ abstract classes and subclasses thereof. C++-level client implementations send messages to local proxies representing individual or group channels (bound via proxy constructor arguments). These proxy objects marshall arguments and transport them through Isis to group members. This use of proxy objects as channels represents one simple and efficient means of tying OID-based C++ constructs to channel-based group constructs.

Clients of services that return results may either collect all replies in an IDL `sequence` or use built-in Isis collation facilities (e.g., choosing only the first response, as is typical in systems using groups primarily for fault-tolerance) that reduce them to a single result.

Group members are linked to dispatchers residing in Isis lightweight process entries. Messages received at entries are unpacked and forwarded to user-defined C++ implementation objects that perform the indicated services. For procedural operations, the dispatchers marshall and send returned results back to clients.

Since IDL does not include constructs to constrain or implement interfaces, all group membership, monitoring, and bookkeeping operations are performed at the C++ level by programmers, mainly via C++ classes and utilities layered above raw C Isis routines.

So far, user applications of RDO/C++ tend to exploit only two strengths of groups, subscription and fault-tolerance. RDO/C++ library classes help automate usage of these common idiomatic constructions. This will include, for example, support for Isis-News, a publish-subscribe tool that is mapped onto model-view-controller[16] style `change` and `update` methods, and also serves as a basis for a CORBA Event Notification Service. Also, RDO/C++ simplifies usage of the Isis *coordinator-cohort* standby protocol in which only one member of a group responds to a service request, backed up by others in case of failure.

Acknowledgements

Thanks to Robert Cooper, Gary Craig, Peter O'Hearn, Marc Shapiro, and Paul Tymann for commenting on drafts on very short notice. The work reported here was supported in part by Isis Distributed Systems, Inc., and the New York CASE Center.

References

[1] Agha, G., & C. Callsen, "ActorSpace: An Open Distributed Programming Paradigm", *Proceedings, Hawaii International Conference on System Sciences*, January, 1993.

[2] Agha, G., I. Mason, S. Smith, & C. Talcott, *A Foundation for Actor Computation*, Technical Report, University of Illinois at Urbana-Champlain, 1993.

[3] Birman, K., "The Process Group Approach to Reliable Distributed Computing", *Communications of the ACM*, December 1993.

[4] Black, A., & M. Immel, "Encapsulating Plurality", *Proceedings, ECOOP '93*, Springer-Verlag, 1993.

[5] Bellur, U., G. Craig, K. Shank, & D. Lea, "Clustering: Composition Methods for Active Object Systems", *Proceedings, Hawaii International Conference on System Sciences*, January 1994.

[6] Casselman, R., *A Role-Based Architectural Model Applied to Object-Oriented Systems*, Thesis, Dept., Systems and Computer Engineering, Carleton University, 1993.

[7] de Champeaux, D., D. Lea., & P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.

[8] Gamma, E., R. Helm, R. Johnson, & J. Vlissides. "Design Patterns: Abstraction and Reuse of Object-Oriented Designs", *Proceedings, ECOOP '93*, Springer-Verlag, 1993.

[9] Gourhant, Y., & Marc Shapiro, "FOG/C++: a Fragmented-Object Generator", *Proceedings, USENIX C++ Conference*, USENIX, 1990.

[10] Hagsand, O., H. Herzog, K. Birman, & R. Cooper, "Object-Oriented Reliable Computing", *Proceedings, International Workshop on Object-Oriented Operating Systems*, IEEE, 1992.

[11] Harrison, W., & H. Ossher, "Subject-Oriented Programming", *Proceedings, OOPSLA '93*, ACM, 1993.

[12] Helm, R., I. Holland, & D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", *Proceedings, OOPSLA '90*, ACM, 1990.

[13] Hogg, J., "Islands: Aliasing Protection In Object-Oriented Languages", *Proceedings, OOPSLA '91*, ACM, 1991.

[14] Hogg, J., D. Lea, R. Holt, A. Wills, & D. de Champeaux, "The Geneva Convention on the Treatment of Object Aliasing", *OOPS Messenger*, April 1992.

[15] Isis Distributed Systems, *ISIS User Guide and Reference Manual*, Isis Distributed Systems, Inc, 111 South Cayuga St., Ithaca NY, 1992.

[16] Krasner, G. & S. Pope, "A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, August/September 1988.

[17] Maffeis, S., "Electra: Making Distributed Programs Object-Oriented", *Proceedings, Symposium on Experiences with Distributed and Multiprocessor Systems*, USENIX, September, 1993.

[18] Makpangou, M., Y. Gourhant, J. Le Narzul, & M. Shapiro "Fragmented Objects for Distributed Abstractions", in *Advances in Distributed Systems*, IEEE, 1993.

[19] Matsuoka, S., T. Watanabe, & A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming", in *Proceedings, ECOOP '91*, Lecture Notes in Computer Science, no 512, Springer Verlag, 1991.

[20] Milner, R., *Communication and Concurrency*, Prentice Hall International, 1989.

[21] Mullender, S. (Ed.) *Distributed Systems*, 2nd ed., Addison-Wesley, 1993.

[22] OMG, *Common Object Request Broker Architecture and Specification*, Document 91.12.1, Object Management Group, 1991.

[23] OMG, *Joint Object Services Submission*, Document 93.2.1, Object Management Group, 1993.

[24] Powell, M., *Objects, References, Identifiers and Equality*, Document 93.7.5, Object Management Group, 1993.

[25] Raj, R., E. Tempero, H. Levy, A. Black, N. Hutchinson, & E. Jul, "Emerald: A General Purpose Programming Language", *Software – Practice and Experience*, 1991.

[26] Scholl, M., C. Laasch, & M. Tresch, "Updatable Views in Object Oriented Databases", in C. Delobel, M. Kifer & Y. Masunaga (eds.) *Deductive and Object-Oriented Databases*, Springer-Verlag, 1991.

[27] Shapiro, M., P. Dickman, & D. Plainfossé, *SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection*, Rapport de Recherche INRIA 1799, 1992.

[28] Strom, R., D. Bacon, A. Goldberg, A. Lowry, D. Yellin, & S. Yemeni, *Hermes: A Language for Distributed Computing*, Prentice Hall, 1991.

[29] Stroustrup, B., *The C++ Programming Language*, 2nd ed., Addison-Wesley, 1991.

[30] Van Renesse, R., K., Birman, R. Cooper, B. Glade, & P. Stephenson, "Reliable Multicast Between Microkernels", *Proceedings, USENIX Workshop on Microkernels and Other Kernel Architectures*, April, 1992.

[31] Wills, A., *Formal Methods Applied to Object Oriented Programming*, Thesis, University of Manchester, 1992.