Applying a Pattern Language to Develop Extensible ORB Middleware

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer Engineering Dept.
University of California, Irvine, USA

Chris Cleeland
cleeland_c@ociweb.com
Object Computing Inc.
St. Louis, MO, USA*

This paper appeared as a chapter in the book *Design Patterns in Communications*, (Linda Rising, ed.), Cambridge University Press, 2000. An abridged version appeared in the IEEE Communications Magazine Special Issue on Patterns (Linda Rising, ed.), Volume 37, No. 4, April 1999.

Abstract

Distributed object computing forms the basis of next-generation communication software. At the heart of distributed object computing are Object Request Brokers (ORBs), which automate many tedious and error-prone distributed programming tasks. Like much communication software, conventional ORBs use statically configured designs, which are hard to port, optimize, and evolve. Likewise, conventional ORBs cannot be extended without modifying their source code, which forces recompilation, relinking, and restarting running ORBs and their associated application objects.

This paper makes two contributions to the study of extensible ORB middleware. First, it presents a case study illustrating how a pattern language can be used to develop dynamically configurable ORBs that can be customized for specific application requirements and system characteristics. Second, we quantify the impact of applying this pattern language to reduce the complexity and improve the maintainability of common ORB tasks, such as connection management, data transfer, demultiplexing, and concurrency control.

1 Introduction

Four trends are shaping the future of commercial software development. First, the software industry is moving away from *programming* applications from scratch to *integrating* applications using reusable components [1]. Second, there is great demand for *distribution technology* that provides remote method invocation and/or message-oriented middleware to simplify application collaboration. Third, there are increasing efforts to define standard software infrastructure frameworks that permit applications to interwork seamlessly throughout *heterogeneous* environments [2]. Finally, next-generation distributed applications such as video-on-demand, teleconferencing, and avionics require *quality-of-service* (QoS) guarantees for latency, bandwidth, and reliability [3].

A key software technology supporting these trends is *distributed object computing (DOC) middleware*. DOC middleware facilitates the collaboration of local and remote application components in heterogeneous distributed environments. The goal of DOC middleware is to eliminate many tedious, error-prone, and non-portable aspects of developing and evolving distributed applications and services. In particular, DOC middleware automates common network programming tasks, such as object location, implementation startup (*i.e.*, server and object activation), encapsulation of byte-ordering and parameter type size differences across dissimilar architectures (*i.e.*, parameter marshaling), fault recovery, and security. At the heart of DOC middleware are *Object Request Brokers* (ORBs), such as CORBA [4], DCOM [5], and Java RMI [6].

This paper describes how we have applied a *pattern language* to develop and evolve dynamically configurable ORB middleware that can be extended more readily than statically configured middleware. In general, pattern languages help to alleviate the continual re-discovery and re-invention of software concepts and components by conveying a family of related solutions to standard software development problems [7]. For instance, pattern languages are useful for documenting the roles and relationships among participants in common communication software architectures [8]. The pattern language presented in this paper is a generalization of the one presented in [9] and has been used successfully to build flexible, efficient, event-driven, and concurrent communication software, including ORB middleware.

To focus our discussion, this paper presents a case study that illustrates how we have applied this pattern language to develop *The ACE ORB* (TAO) [10]. TAO is a freely available, highly extensible ORB targeted for applications with real-time QoS requirements, including avionics mission computing [11], multimedia applications [12], and distributed interactive simulations [13]. A novel aspect of TAO is its extensible design, which is guided by a pattern language that enables the ORB to be customized dynamically to meet specific application QoS requirements and network/endsystem characteristics.

The remainder of this paper is organized as follows: Sec-

*This work was supported in part by ATD, BBN, Boeing, Cisco, DARPA contract 9701516, Motorola Commercial Government and Industrial Solutions Sector, Motorola Laboratories, Siemens, and Sprint.

tion 2 presents an overview of CORBA and TAO; Section 3 motivates the need for dynamic configuration and describes the pattern language that resolves key design challenges faced when developing extensible ORBs; Section 3.5 evaluates and quantifies the contribution of the pattern language to ORB middleware; and Section 4 presents concluding remarks.

2 Overview of CORBA and TAO

This section outlines the CORBA reference model and describes the enhancements that TAO provides for high-performance and real-time applications.

2.1 Overview of the CORBA Reference Model

CORBA Object Request Brokers (ORBs) [14] allow clients to invoke operations on distributed objects without concern for the following issues:

Object location: A CORBA object either can be collocated with the client or distributed on a remote server, without affecting its implementation or use.

Programming language: The languages supported by CORBA include C, C++, Java, Ada95, COBOL, and Smalltalk, among others.

OS platform: CORBA runs on many OS platforms, including Win32, UNIX, MVS, and real-time embedded systems, such as VxWorks, Chorus, and LynxOS.

Communication protocols and interconnects: The communication protocols and interconnects that CORBA run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, embedded system backplanes, and shared memory.

Hardware: CORBA shields applications from side effects stemming from hardware diversity, such as different storage layouts and data type sizes/ranges.

Figure 1 illustrates the components in the CORBA reference model, all of which collaborate to provide the portability, interoperability and transparency outlined above.

Each component in the CORBA reference model is outlined below:

Client: A client is a *role* that obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Clients can access remote objects just like a local object, *i.e.*, `object`$\rightarrow$`operation(args)`. Figure 1 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

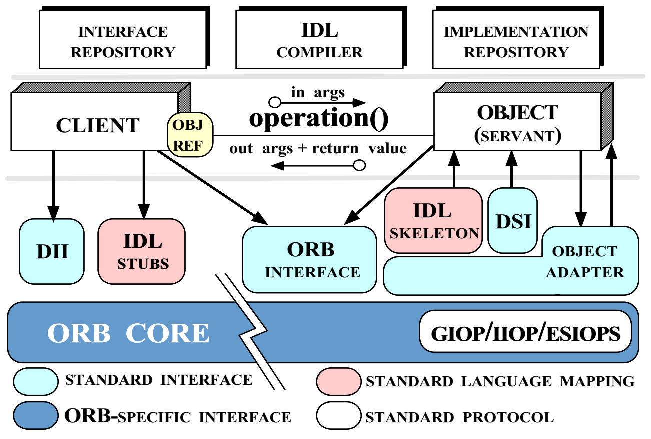


Figure 1: Components in the CORBA Reference Model

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *object ID* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it to implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `struct`s. A client never interacts with servants directly, but always through objects identified by object references. Together, an object and its servant form an implementation of the Bridge pattern [15], with *object* as the *RefinedAbstraction* and *servant* as the *ConcreteImplementor*.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined [16].

ORB Interface: An ORB is an abstraction that can be implemented various ways, *e.g.*, one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an interface to an ORB. This ORB interface provides standard operations to initialize and shut down the ORB, convert object references to strings and back, and create argument lists for requests made through the *dynamic invocation interface* (DII).

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [15] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern [15] and demarshal the message-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [17].

Dynamic Invocation Interface (DII): The DII allows clients to generate requests at run-time, which is useful when an application has no compile-time knowledge of the interface it accesses. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of two-way operations to avoid blocking the client until the servant responds. CORBA SII stubs support both synchronous and asynchronous *two-way*, *i.e.*, request/response and *one-way*, *i.e.*, request-only operations.

Dynamic Skeleton Interface (DSI): The DSI is the server's analogue to the client's DII. The DSI allows an ORB to deliver requests to servants that have no compile-time knowledge of the IDL interface they implement. Clients making requests need not know whether the server ORB uses static skeletons or dynamic skeletons. Likewise, servers need not know if clients use the DII or SII to invoke requests.

Object Adapter: An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

Interface Repository: The Interface Repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet be able to determine what operations are valid on the object and make invocations on it using the DII. In addition, the Interface Repository provides a common location to store additional information associated with interfaces to CORBA objects, such as type libraries for stubs and skeletons.

Implementation Repository: The Implementation Repository [18] contains information that allows an ORB to activate servers to process servants. Most of the information in the Implementation Repository is specific to an ORB or OS environment. In addition, the Implementation Repository provides a common location to store information associated with servers, such as administrative control, resource allocation, security, and activation modes.

2.2 Overview of TAO

TAO is a high-performance, real-time ORB endsystem targeted for applications with deterministic and statistical QoS requirements, as well as best-effort requirements. TAO's ORB endsystem contains the network interface, OS, communication protocol, and CORBA-compliant middleware components and services shown in Figure 2. TAO supports the standard OMG

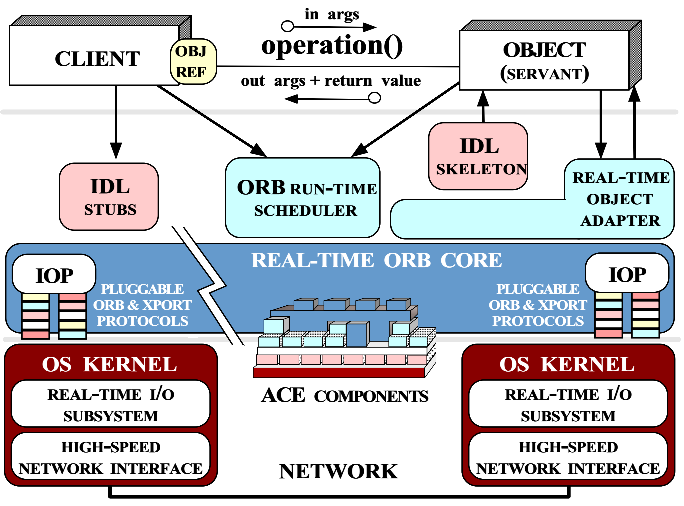


Figure 2: Components in the TAO Real-time ORB Endsystem

CORBA reference model [14] and Real-time CORBA specification [19], with enhancements designed to ensure efficient, predictable, and scalable QoS behavior for high-performance and real-time applications. In addition, TAO is well-suited for general-purpose distributed applications. Below, we outline the features of TAO's components shown in Figure 2.

Optimized IDL Stubs and Skeletons: IDL stubs and skeletons perform marshaling and demarshaling of application operation parameters, respectively. TAO's IDL compiler generates stubs/skeletons that can selectively use highly optimized compiled and/or interpretive (de)marshaling [20]. This flexibility allows application developers to selectively trade off time and space, which is crucial for high-performance, real-time, and/or embedded distributed systems.

Real-time Object Adapter: An Object Adapter associates servants with the ORB and demultiplexes incoming requests to servants. TAO's real-time Object Adapter uses perfect hashing [21] and active demultiplexing [22] optimizations to dispatch servant operations in constant $O(1)$ time, regardless of the number of active connections, servants, and operations defined in IDL interfaces.

Run-time Scheduler: TAO's run-time scheduler [19] maps application QoS requirements, such as bounding end-to-end latency and meeting periodic scheduling deadlines, to ORB endsystem/network resources, such as CPU, memory, network connections, and storage devices. TAO's run-time scheduler supports both static [10] and dynamic [23] real-time scheduling strategies.

Real-time ORB Core: An ORB Core delivers client requests to the Object Adapter and returns responses (if any) to clients. TAO's real-time ORB Core [24] uses a multi-threaded, preemptive, priority-based connection and concurrency architecture [20] to provide an efficient and predictable CORBA protocol engine. TAO's ORB Core allows customized protocols to be plugged into the ORB without affecting the standard CORBA application programming model [25].

Real-time I/O subsystem: TAO's real-time I/O (RIO) subsystem [26] extends support for CORBA into the OS. RIO assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be enforced. When integrated with advanced hardware, such as the high-speed network interfaces described below, RIO can (1) perform early demultiplexing of I/O events onto prioritized kernel threads to avoid thread-based priority inversion and (2) maintain distinct priority streams to avoid packet-based priority inversion. TAO also runs efficiently and as predictably as possible on conventional I/O subsystems that lack advanced QoS features.

High-speed network interface: At the core of TAO's I/O subsystem is a "daisy-chained" network interface consisting of one or more ATM Port Interconnect Controller (APIC) chips [27]. The APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps using zero-copy buffering optimization to avoid data copying across endsystem layers. In addition, TAO runs on conventional real-time interconnects, such as VME backplanes and multi-processor shared memory environments, as well as TCP/IP.

TAO internals: TAO is developed using lower-level middleware called ACE [28], which implements core concurrency and distribution patterns [8] for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications and higher-level middleware like TAO. ACE and TAO run on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems, such as Sun/Chorus ClassiX, LynxOS, and VxWorks.

To expedite our project goals, and to avoid re-inventing existing components, we based TAO on SunSoft IIOP, which is a freely available C++ reference implementation of the Internet Inter-ORB Protocol (IIOP) version 1.0. Although SunSoft IIOP provides core features of a CORBA ORB it also has the following limitations:

Lack of standard ORB features: Although SunSoft IIOP provides an ORB Core, an IIOP 1.0 protocol engine, and a DII and DSI implementation, it lacks an IDL compiler, an Interface Repository and Implementation Repository, and a Portable Object Adapter (POA). TAO implements all these missing features and provides newer CORBA features, asynchronous method invocations [29], real-time CORBA [19] features [30], and fault tolerance CORBA features [31, 32].

Lack of IIOP optimizations: Due to the excessive marshaling/demarshaling overhead, data copying, and high-levels of function call overhead, SunSoft IIOP performs poorly over high-speed networks. Therefore, we applied a range of optimization principle patterns [22] that improved its performance considerably [33]. The principles that directed our optimizations include: (1) optimizing for the common case, (2) eliminating gratuitous waste, (3) replacing general-purpose methods with efficient special-purpose ones, (4) precomputing values, if possible, (5) storing redundant state to speed up expensive operations, (6) passing information between layers, and (7) optimizing for processor cache affinity.

This paper does not discuss how TAO solves the limitations with SunSoft IIOP outlined above, which are described in detail in [10, 20]. Instead, we focus on how TAO uses patterns to implement an ORB that overcomes the following SunSoft IIOP limitations while simultaneously preserving its QoS capabilities:

Lack of portability: Like most communication software, SunSoft IIOP is programmed directly using low-level networking and OS APIs, such as sockets, `select`, and POSIX Pthreads. Not only are these APIs tedious and error-prone, they are also not portable across OS platforms, *e.g.,* many operating systems lack Pthreads support. Section 3.3.1 illustrates how we used the *Wrapper Facade* pattern [15] to improve TAO's portability.

Lack of configurability: Like many ORBs and other middleware, SunSoft IIOP is configured *statically*, which makes it hard to extend without modifying its source code directly.

This violated a key design goal of TAO, namely *dynamic* adaptation to diverse application requirements and system environments. Sections 3.3.7, 3.3.6, and 3.3.8 explain how we used the *Abstract Factory* [15], *Strategy* [15], and *Component Configurator* [8] patterns to simplify the TAO's configurability for different use-cases.

Lack of software cohesion: Like many applications, SunSoft IIOP focuses on solving a specific problem, *i.e.*, implementing an ORB Core and an IIOP protocol engine. It accomplish this using a tightly-coupled, *ad-hoc* implementation that hard-codes key ORB design decisions. Sections 3.3.7 and 3.3.6 explain how we used *Abstract Factory* and *Strategy* to decrease the unnecessary coupling and increase cohesion when evolving SunSoft IIOP to TAO.

3 Applying a Pattern Language to Build Extensible ORB Middleware

3.1 Why We Need Dynamically Configurable Middleware

A key motivation for ORB middleware is to offload complex, lower-level distributed system infrastructure tasks from application developers to ORB developers. ORB developers are responsible for implementing reusable middleware components that handle connection management, interprocess communication, concurrency, transport endpoint demultiplexing, scheduling, dispatching, (de)marshaling, and error handling. These components are typically compiled into a run-time ORB library, linked with application objects that use the ORB components, and executed in one or more OS processes.

Although this separation of concerns can simplify application development, it can also yield inflexible and inefficient applications and middleware architectures. The primary reason is that many conventional ORBs are configured *statically* at compile-time and link-time by ORB developers, rather than *dynamically* at installation-time or run-time by application developers. Statically configured ORBs have the following drawbacks [28]:

Inflexibility: Statically-configured ORBs tightly couple each component's *implementation* with the *configuration* of internal ORB components, *i.e.*, which components work together and how they work together. As a result, extending statically-configured ORBs requires modifications to existing source code. In commercial non-open-source ORBs, this code may not be accessible to application developers.

Even if source code is available, extending statically-configured ORBs requires recompilation and relinking. Moreover, any currently executing ORBs and their associated objects must be shutdown and restarted. This static reconfiguration process is not well-suited for application domains, such as telecom call processing, that require 7×24 availability [34].

Inefficiency: Statically-configured ORBs can be inefficient, both in terms of space and time. Space inefficiency can occur if unnecessary components are always statically configured into an ORB. This can increase the ORB's memory footprint, forcing applications to pay a space penalty for features they do not require. Overly large memory footprints are particularly problematic for embedded systems, such as cellular phones or telecom switch line cards [35].

Time inefficiency can stem from restricting an ORB to use statically configured algorithms or data structures for key processing tasks, thereby making it hard for application developers to customize an ORB to handle new use-cases. For instance, real-time avionics systems [11] often can instantiate all their servants off-line. These systems can benefit from an ORB that uses perfect hashing or active demultiplexing [36] to demultiplex incoming requests to servants. Thus, ORBs that are configured statically to use a general-purpose, "one-size-fits-all" demultiplex *strategy*, such as dynamic hashing, may perform poorly for mission-critical systems.

In theory, the drawbacks with static configuration described above are *internal* to ORBs and should not affect application developers directly. In practice, however, application developers are inevitably affected since the quality, portability, usability, and performance of the ORB middleware is reduced. Therefore, an effective way to improve ORB extensibility is to develop ORB middleware that can be both statically *and* dynamically configured.

Dynamic configuration enables the selective integration of customized implementations for key ORB strategies, such as connection management, communication, concurrency, demultiplexing, scheduling, and dispatching. This design allows ORB developers to concentrate on the *functionality* of ORB components, without committing themselves prematurely to a specific *configuration* of these components. Moreover, dynamic configuration enables application developers and ORB developers to change design decisions late in the system lifecycle, *i.e.*, at installation-time or run-time.

Figure 3 illustrates the following key dimensions of ORB extensibility:

1. Extensibility to retarget the ORB on new platforms, which requires that the ORB be implemented using modular components that shield it from non-portable system mechanisms, such as those for threading, communication, and event demultiplexing. OS platforms such as POSIX, Win32, VxWorks, and MVS provide a wide variety of system mechanisms.

2. Extensibility via custom implementation strategies, which can be tailored to specific application requirements. For

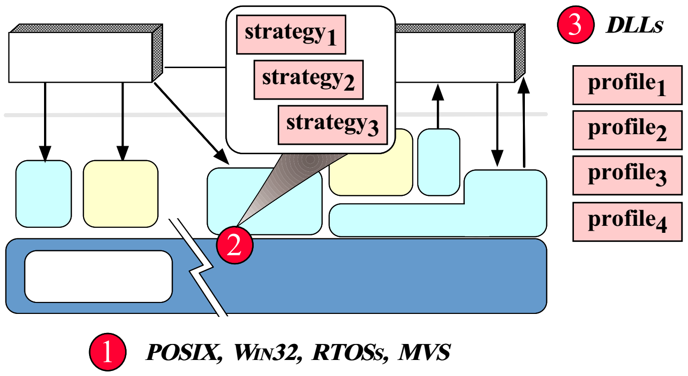


Figure 3: Dimensions of ORB Extensibility

instance, ORB components can be customized to meet periodic deadlines in real-time systems [11]. Likewise, ORB components can be customized to account for particular system characteristics, such as the availability of asynchronous I/O or high-speed ATM networks.

3. Extensibility via dynamic configuration of custom strategies, which takes customization to the next level by dynamically linking only those strategies that are necessary for a specific ORB "personality." For example, different application domains, such as medical systems or telecom call processing, may require custom combinations of concurrency, scheduling, or dispatch strategies. Configuring these strategies at run-time from dynamically linked libraries (DLLs) can (1) reduce the memory footprint of an ORB and (2) make it possible for application developers to extend the ORB without requiring access or changes to the original source code.

Below, we describe the pattern language applied to enhance the extensibility of TAO along each dimension outlined above.

3.2 Overview of a Pattern Language that Improves ORB Extensibility

This section uses TAO as a case study to illustrate a pattern language that can help developers of applications and ORBs build, maintain, and extend communication software by reducing the coupling between components. Figure 4 shows the patterns in the pattern language that we applied to develop an extensible ORB architecture for TAO. It is beyond the scope of this paper to describe each pattern in detail or to discuss all the patterns used within TAO. Instead, we focus on how key patterns can improve the extensibility and performance of real-time ORB middleware. The references in [9, 15] contain comprehensive descriptions of these patterns and [8] explains how the patterns can be woven together to form a pattern language.

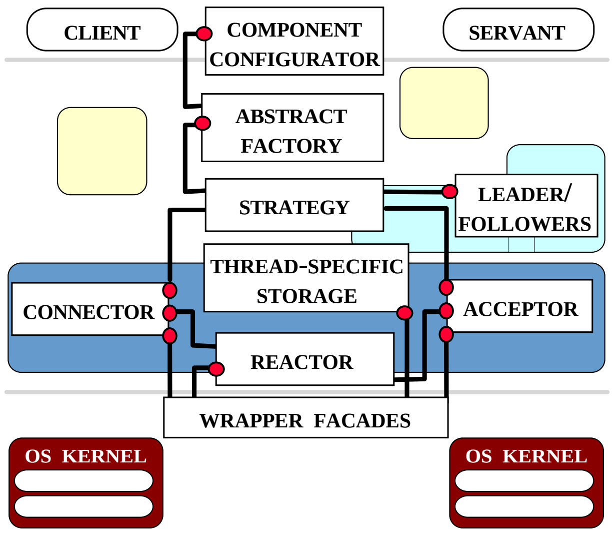


Figure 4: Applying a Pattern Language to TAO

The intent and usage of the patterns in this language are outlined below:

Wrapper Facade [8]: This pattern encapsulates the functions and data provided by existing non-OO APIs within more concise, robust, portable, maintainable, and cohesive OO class interfaces. TAO uses this pattern to avoid tedious, non-portable, and non-typesafe programming of low-level, OS-specific system calls, such as the Socket API or POSIX threads.

Reactor [8]: This pattern structures event-driven applications, particularly servers, that receive requests from multiple clients concurrently but process them iteratively. TAO uses this pattern to notify ORB-specific handlers synchronously when I/O events occur in the OS. The Reactor pattern drives the main event loop in TAO's ORB Core, which accepts connections and receives/sends client requests/responses.

Acceptor-Connector [8]: This pattern decouples connection establishment and service initialization from service processing in a networked system. TAO uses this pattern in the ORB Core on servers and clients to passively and actively establish GIOP connections that are independent of the underlying transport mechanisms.

Leader/Followers [8]: This pattern provides an efficient concurrency model in which multiple threads take turns to share a set of event sources to detect, demultiplex, dispatch and process service requests that occur on the event sources. TAO uses this pattern uses this pattern to facilitate the use of multiple concurrency strategies that can be configured flexibly into its ORB Core at run-time.

Thread-Specific Storage [8]: This pattern allows multiple threads to use a "logically global" access point to retrieve an object that is local to a thread, without incurring locking overhead for each access to the object. TAO uses this pattern to minimize lock contention and priority inversion for real-time applications.

Strategy [15]: This pattern provides an abstraction for selecting one of several candidate algorithms and packaging it into an object. TAO uses this pattern throughout its software architecture to extensibly configure custom ORB strategies for concurrency, communication, scheduling, and demultiplexing.

Abstract Factory [15]: This pattern provides a single component that builds related objects. TAO uses this pattern to consolidate its dozens of *Strategy* objects into a manageable number of abstract factories that can be reconfigured *en masse* into clients and servers conveniently and consistently. TAO components use these factories to access related strategies without specifying their subclass name explicitly.

Component Configurator [8]: This pattern allows an application to link and unlink its component implementations at run-time without having to modify, recompile or statically relink the application. It also supports the reconfiguration of components into different processes without having to shut down and re-start running processes. TAO uses this pattern to dynamically interchange *abstract factory* implementations in order to customize ORB personalities at run-time.

The patterns constituting this pattern language are not limited to ORBs or communication middleware. They have been applied in many other communication application domains, including telecom call processing and switching, avionics flight control systems, multimedia teleconferencing, and distributed interactive simulations.

3.3 How to Use a Pattern Language to Resolve ORB Design Challenges

In the following discussion, we outline the forces underlying the key design challenges that arise when developing extensible real-time ORBs. We also describe which pattern(s) in our pattern language resolve these forces and explain how these patterns are used in TAO. In addition, we show how the absence of these patterns in an ORB leaves these forces unresolved. To illustrate this latter point concretely, we compare TAO with SunSoft IIOP. Since TAO evolved from the SunSoft IIOP release, it provides an ideal baseline to evaluate the impact of patterns on the software qualities of ORB middleware.

3.3.1 Encapsulate Low-level System Mechanisms with the *Wrapper Facade* Pattern

Context: One role of an ORB is to shield application-specific clients and servants from the details of low-level systems programming. Thus, ORB developers, rather than application developers, are responsible for tedious, low-level network programming tasks, such as demultiplexing events, sending and receiving GIOP messages across the network, and spawning threads to execute client requests concurrently. Figure 5 illustrates a common approach used by SunSoft IIOP,

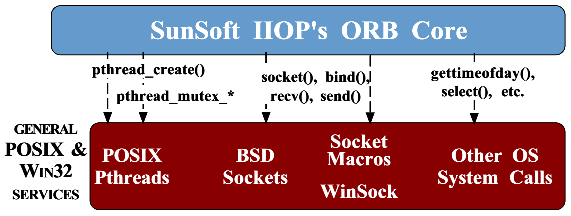


Figure 5: SunSoft IIOP Operating System Interaction

which is programmed internally using system mechanisms, such as sockets, `select`, and POSIX threads.

Problem: Developing an ORB is hard. It is even harder if developers must wrestle with low-level system mechanisms written in languages like C, which often yield the following problems:

• ***ORB developers must have intimate knowledge of many OS platforms*:** Implementing an ORB using system-level C APIs forces developers to deal with non-portable, tedious, and error-prone OS idiosyncrasies, such as using untyped socket handles to identify transport endpoints. Moreover, these APIs are not portable across OS platforms. For example, Win32 lacks POSIX Pthreads and has subtly different semantics for sockets and `select`.

• ***Increased maintenance effort*:** One way to build an ORB is to handle portability variations via explicit conditional compilation directives in ORB source code. However, using conditional compilation to address platform-specific variations *at all points of use* increases the complexity of the source code, as shown in Section 3.5. Extending such ORBs is hard since platform-specific details are scattered throughout the implementation source code files.

• ***Inconsistent programming paradigms*:** System mechanisms are accessed through C-style function calls, which cause an "impedance mismatch" with the OO programming style supported by C++, the language we use to implement TAO.

How can we avoid accessing low-level system mechanisms when implementing an ORB?

Solution $\rightarrow$ the Wrapper Facade pattern: An effective way to avoid accessing system mechanisms directly is to use the *Wrapper Facade* pattern [8], which is a variant of the Facade pattern [15]. The intent of the Facade pattern is to simplify the interface for a subsystem. The intent of the Wrapper Facade pattern is more specific: it provides typesafe, modular, and portable OO interfaces that encapsulate lower-level, stand-alone system mechanisms, such as sockets, `select`, and POSIX threads. In general, the Wrapper Facade pattern should be applied when existing system-level APIs are non-portable and non-typesafe.

Using the Wrapper Facade pattern in TAO: TAO accesses all system mechanisms via the wrapper facades provided by ACE [28]. Figure 6 illustrates how the ACE C++ wrapper facades improve TAO's robustness and portability by encapsulating and enhancing native OS concurrency, communication, memory management, event demultiplexing, and dynamic linking mechanisms with typesafe OO interfaces. The

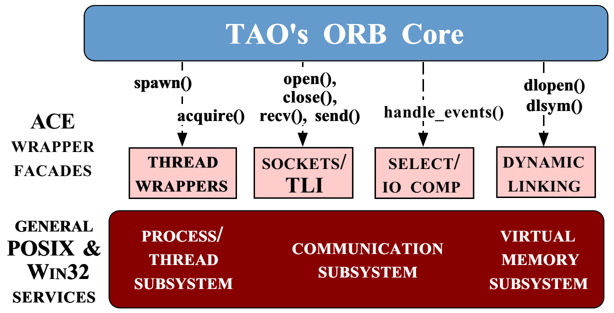


Figure 6: Using the Wrapper Facade Pattern to Encapsulate Native OS Mechanisms

OO encapsulation provided by ACE alleviates the need for TAO to access weakly-typed system APIs directly. Thus, C++ compilers can detect type system violations at compile-time rather than waiting for the problems to occur at run-time.

The ACE wrapper facades use C++ features to eliminate performance penalties that would otherwise be incurred from its additional type safety and layer of abstraction. For instance, inlining is used to avoid the overhead of calling small methods. Likewise, static methods are used to avoid the overhead of passing a C++ `this` pointer to each invocation.

Although the ACE wrapper facades resolve several common low-level development problems, they are just the first step towards developing an extensible ORB. The remaining patterns described in this section build on the encapsulation provided by the ACE wrapper facades to address more challenging ORB design issues.

3.3.2 Demultiplexing ORB Core Events Using the Reactor Pattern

Context: An ORB Core is responsible for demultiplexing I/O events from multiple clients and dispatching their associated event handlers. For instance, a server-side ORB Core listens for new client connections and reads/writes GIOP requests/responses from/to connected clients. To ensure responsiveness to multiple clients, an ORB Core uses OS event demultiplexing mechanisms to wait for CONNECTION, READ, and WRITE events to occur on multiple socket handles. Common event demultiplexing mechanisms include `select`, `WaitForMultipleObjects`, I/O completion ports, and threads.

Figure 7 illustrates a typical event demultiplexing sequence for SunSoft IIOP. In (**1**), the server enters its event

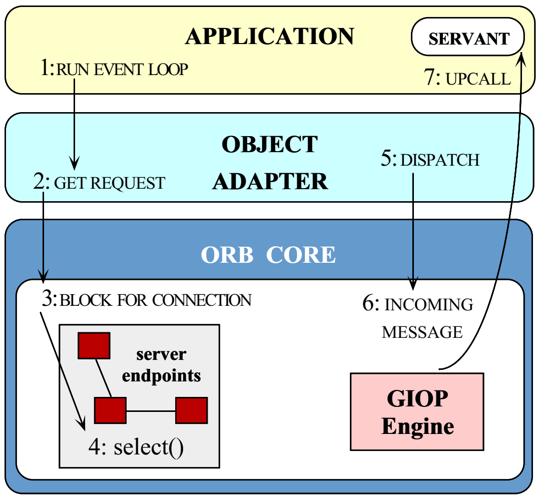


Figure 7: The SunSoft IIOP Event Loop

loop by (**2**) calling `get_request` on the Object Adapter. The `get_request` method then (**3**) calls the static method `block_for_connection` on the `server_endpoint`. This method manages all aspects of server-side connection management, ranging from connection establishment to GIOP protocol handling. The ORB remains blocked (**4**) on `select` until the occurrence of I/O event, such as a connection event or a request event. When a request event occurs, `block_for_connection` demultiplexes that request to a specific `server_endpoint` and (**5**) dispatches the event to that endpoint. The GIOP Engine in the ORB Core then (**6**) retrieves data from the socket and passes it to the Object Adapter, which demultiplexes it, demarshals it, and (**7**) dispatches the appropriate method upcall to the user-supplied servant.

Problem: One way to develop an ORB Core is to hard-code it to use one event demultiplexing mechanism, such as `select`. Relying on just one mechanism is undesirable, however, since no single scheme is efficient on all platforms or for all application requirements. For instance, asynchronous I/O completion ports are highly efficient on Windows NT [37], whereas synchronous threads are an efficient demultiplexing mechanism on Solaris [33].

Another way to develop an ORB Core is to tightly couple its event demultiplexing code with the code that performs GIOP protocol processing. For instance, the event demultiplexing logic of SunSoft IIOP is not a self-contained component. Instead, it is closely intertwined with subsequent processing of client request events by the Object Adapter and IDL skeletons. In this case, however, the demultiplexing code cannot be reused as a blackbox component by other communication middleware applications, such as HTTP servers [37] or video-on-demand servers. Moreover, if new ORB strategies for threading or Object Adapter request scheduling algorithms are introduced, substantial portions of the ORB Core must be rewritten.

How then can an ORB implementation decouple itself from a specific event demultiplexing mechanism and decouple its demultiplexing code from its handling code?

Solution → the Reactor pattern: An effective way to reduce coupling and increase the extensibility of an ORB Core is to apply the *Reactor* pattern [8]. This pattern supports synchronous demultiplexing and dispatching of multiple *event handlers*, which are triggered by events that can arrive concurrently from multiple sources. The Reactor pattern simplifies event-driven applications by integrating the demultiplexing of events and the dispatching of their corresponding event handlers. In general, the Reactor pattern should be applied when applications or components, such as an ORB Core, must handle events from multiple clients concurrently, without becoming tightly coupled to a single low-level mechanism, such as `select`.

Note that applying the Wrapper Facade pattern is not sufficient to resolve the event demultiplexing problems outlined above. A wrapper facade for `select` may improve ORB Core portability somewhat. However, this pattern alone does not resolve the need to completely decouple the low-level event demultiplexing logic from the higher-level client request processing logic in an ORB Core. Recognizing the limitations of the Wrapper Facade pattern, and then applying the Reactor pattern to overcome the limitations, is one of the benefits of applying a pattern language, rather than just isolated patterns.

Using the Reactor pattern in TAO: TAO uses the Reactor pattern to drive the main event loop in its ORB Core, as shown in Figure 8. A TAO server (**1**) initiates an event loop in the ORB Core's *`Reactor`*, where it (**2**) remains blocked on `select` until an I/O event occurs. When a GIOP request event occurs, the `Reactor` demultiplexes the request to the appropriate event handler, which is the GIOP `Connection_Handler` that is associated with each connected socket. The `Reactor` (**3**) then calls `Connection_Handler::handle_input`, which (**4**) dispatches the request to TAO's Object Adapter. The Object Adapter demultiplexes the request to the appropriate upcall method on the servant and (**5**) dispatches the upcall.

The Reactor pattern enhances the extensibility of TAO by decoupling the event handling portions of its ORB Core from the underlying OS event demultiplexing mechanisms. For example, the `WaitForMultipleObjects`

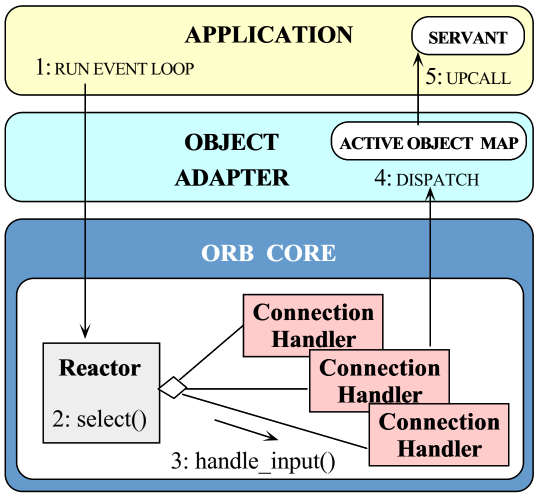


Figure 8: Using the Reactor Pattern in TAO's Event Loop

event demultiplexing system call can be used on Windows NT, whereas `select` can be used on UNIX platforms. Moreover, the Reactor pattern simplifies the configuration of new event handlers. For instance, adding a new `Secure_Connection_Handler` that performs encryption/decryption of all network traffic will not affect the `Reactor`'s implementation. Finally, unlike the event demultiplexing code in SunSoft IIOP, which is tightly coupled to one use-case, the ACE implementation of the Reactor pattern [8] used by TAO has been applied in many other OO event-driven applications ranging from HTTP servers [37] to real-time avionics infrastructure [11].

3.3.3 Managing Connections in an ORB Using the Acceptor-Connector Pattern

Context: Managing connections is another key responsibility of an ORB Core. For instance, an ORB Core that implements the IIOP protocol must establish TCP connections and initialize the protocol handlers for each IIOP `server_endpoint`. By localizing connection management logic in the ORB Core, application-specific servants can focus solely on processing client requests, rather than dealing with low-level network programming tasks.

An ORB Core is not *limited* to running over IIOP and TCP transports, however. For instance, while TCP can transfer GIOP requests reliably, its flow control and congestion control algorithms can preclude its use as a real-time protocol [10]. Likewise, it may be more efficient to use a shared memory transport mechanism when clients and servants are collocated on the same endsystem. Thus, an ORB Core should be flexible enough to support multiple transport mechanisms [16].

Problem: The CORBA architecture explicitly decouples (1) the connection management tasks performed by an ORB Core from (2) the request processing performed by application-specific servants. However, a common way to implement an ORB's *internal* connection management activities is to use low-level network APIs, such as Sockets. Likewise, the ORB's connection establishment protocol is often tightly coupled with its communication protocol.

For example, Figure 9 illustrates the connection management structure of SunSoft IIOP. The client-side of SunSoft

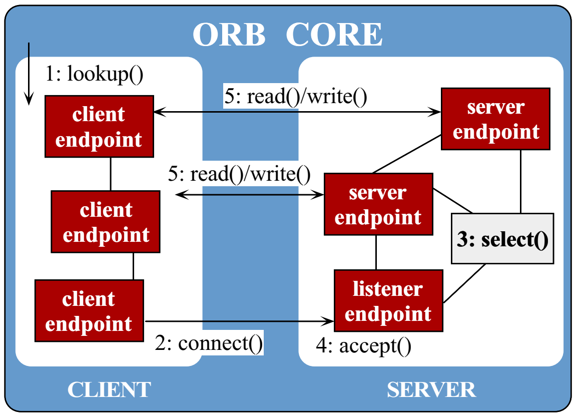


Figure 9: Connection Management in SunSoft IIOP

IIOP implements a hard-coded connection caching *strategy* that uses a linked-list of `client_endpoint` objects. As shown in Figure 9, this list is traversed to find an unused endpoint whenever (**1**) `client_endpoint::lookup` is called. If no unused `client_endpoint` to the server is in the cache, a new connection (**2**) is initiated; otherwise an existing connection is reused. Likewise, the server-side uses a linked list of `server_endpoint` objects to generate the read/write bitmasks required by the (**3**) `select` event demultiplexing mechanism. This list maintains passive transport endpoints that (**4**) accept connections and (**5**) receive requests from clients connected to the server.

The problem with SunSoft IIOP's design is that it tightly couples (1) the ORB's connection management implementation with the socket network programming API and (2) the TCP/IP connection establishment protocol with the GIOP communication protocol, thereby yielding the following drawbacks:

• **Inflexibility:** If an ORB's connection management data structures and algorithms are too closely intertwined, substantial effort is required to modify the ORB Core. For instance, tightly coupling the ORB to use the Socket API makes it hard to change the underlying transport mechanism, *e.g.*, to use shared memory rather than Sockets. Thus, it can be hard to port such a tightly coupled ORB Core to new communication mechanisms, such as ATM, Fibrechannel, or shared memory, or different network programming APIs, such as TLI or Win32 Named Pipes.

• **Inefficiency:** Many internal ORB strategies can be optimized by allowing both ORB developers and application developers to select appropriate implementations late in the software development cycle, *e.g.*, after systematic performance profiling. For example, to reduce lock contention and overhead, a multi-threaded, real-time ORB client may need to store transport endpoints in thread-specific storage [8]. Similarly, the concurrency strategy for a CORBA server might require that each connection run in its own thread to eliminate per-request locking overhead. If connection management mechanisms are hard-coded and tightly bound with other internal ORB strategies, however, it is hard to accommodate efficient new strategies.

How then can an ORB Core's connection management components support multiple transports and allow connection-related behaviors to be (re)configured flexibly late in the development cycle?

Solution → the Acceptor-Connector pattern: An effective way to increase the flexibility of ORB Core connection management and initialization is to apply the *Acceptor-Connector* pattern [8]. This pattern decouples connection initialization from the processing performed after a connection endpoint is initialized. The `Acceptor` component in this pattern is responsible for *passive* initialization, *i.e.*, the server-side of the ORB Core. Conversely, the `Connector` component in the pattern is responsible for *active* initialization, *i.e.*, the client-side of the ORB Core. In general, the Acceptor-Connector pattern should be applied when client/server middleware must allow flexible configuration of network programming APIs and must maintain proper separation of initialization roles.

Using the Acceptor-Connector pattern in TAO: TAO uses the Acceptor-Connector pattern in conjunction with the Reactor pattern to handle connection establishment for GIOP/IIOP communication. Within TAO's client-side ORB Core, a `Connector` initiates connections to servers in response to an operation invocation or an explicit binding to a remote object. Within TAO's server-side ORB Core, an `Acceptor` creates a GIOP `Connection Handler` to service each new client connection. `Acceptors` and `Connection_Handlers` both derive from an `Event_Handler`, which enable them to be dispatched automatically by a `Reactor`.

TAO's `Acceptors` and `Connectors` can be configured with any transport mechanisms, such as Sockets or TLI, provided by the ACE wrapper facades. In addition, TAO's `Acceptor` and `Connector` can be imbued with custom strategies to select an appropriate concurrency mechanism, as described in Section 3.3.4.

Figure 10 illustrates the use of *Acceptor-Connector* strategies in TAO's ORB Core. When a client (**1**) invokes a remote operation, it makes a `connect` call through a `Strategy_Connector`. This `Strategy_Connector`

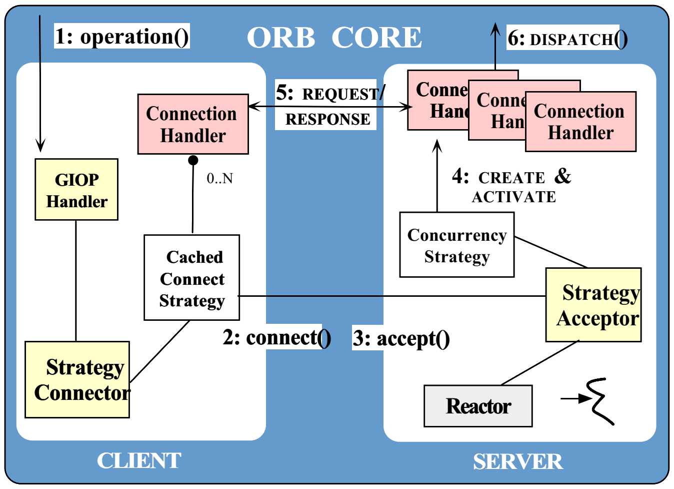


Figure 10: Using the Acceptor-Connector Pattern in TAO's Connection Management

(**2**) consults its *connection strategy* to obtain a connection. In this example, the client uses a "caching connection strategy" that recycles connections to the server and only creates new connections when existing connections are all busy. This caching strategy minimizes connection setup time, thereby reducing end-to-end request latency.

In the server-side ORB Core, the `Reactor` notifies TAO's `Strategy_Acceptor` to (**3**) accept newly connected clients and create `Connection_Handlers`. The `Strategy_Acceptor` delegates the choice of concurrency mechanism to one of TAO's *concurrency* strategies, *e.g.*, reactive, thread-per-connection, or thread-per-priority, described in Section 3.3.4. After a `Connection_Handler` is activated (**4**) within the ORB Core, it performs the requisite GIOP protocol processing (**5**) on a connection and ultimately dispatches (**6**) the request to the appropriate servant via TAO's Object Adapter.

3.3.4 Simplifying ORB Concurrency Using the Leader/Followers Pattern

Context: After the Object Adapter has dispatched a client request to the appropriate servant, the servant executes the request. Execution may occur in the same thread of control as the `Connection_Handler` that received it. Conversely, execution may occur in a different thread, concurrent with other request executions.

The Real-time CORBA specification [38] defines a thread pool API. In addition, the CORBA specification defines an interface on the POA for an application to specify that all requests be handled by a single thread or be handled using an ORB's internal multi-threading policy. To meet application QoS requirements, it is important to develop ORBs that implement these various concurrency APIs efficiently [24]. Concurrency allows long-running operations to execute simultaneously without impeding the progress of other operations. Likewise, preemptive multi-threading is crucial to minimize the dispatch latency of real-time systems [11].

Concurrency is often implemented via the multi-threading capabilities available on OS platforms. For instance, SunSoft IIOP supports the two concurrency architectures shown in Figure 11: a single-threaded Reactive architecture and a thread-per-connection architecture. SunSoft IIOP's reactive concurrency architecture uses `select` within a single thread to dispatch each arriving request to an individual `server_endpoint` object, which subsequently reads the request from the appropriate OS kernel queue. In (**1**), a request arrives and is queued by the OS. Then, `select` fires, (**2**) notifying the associated `server_endpoint` of a waiting request. The `server_endpoint` finally (**3**) reads the request from the queue and processes it.

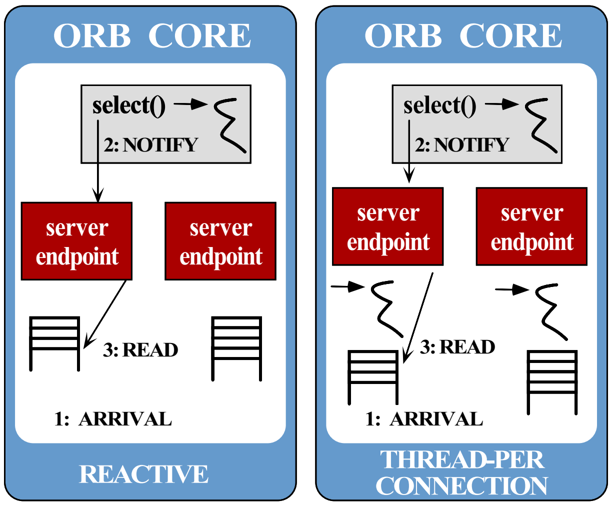


Figure 11: SunSoft IIOP Concurrency Architectures

In contrast, SunSoft IIOP's thread-per-connection architecture executes each `server_endpoint` in its own thread of control, servicing all requests arriving on that connection within its thread. After a connection is established, `select` waits for events on the connection's descriptor. When (**1**) requests are received by the OS, the thread performing `select` (**2**) reads one from the queue and (**3**) hands it off to a `server_endpoint` for processing.

Problem: In many ORBs, the concurrency architecture is programmed directly using the OS platform's multi-threading API, such as the POSIX threads API [39]. However, there are several drawbacks to this approach:

- ***Non-portable*:** Threading APIs are highly platform-specific. Even IEEE standards, such as POSIX threads [39], are not available on many widely-used OS platforms, including Win32, VxWorks, and pSoS. Not only is there no direct syntactic mapping between APIs, but there is no clear mapping of semantics either. For instance, POSIX threads support deferred thread cancellation, whereas Win32 threads do not. Moreover, although Win32 has a thread termination API, the Win32 documentation strongly recommends *not* using it since it does not release all thread resources after a thread exits. Moreover, even POSIX Pthread implementations are non-portable since many UNIX vendors support different drafts of the Pthreads specification.

• ***Hard to program correctly*:** Portability aside, programming a multi-threaded ORB is hard since application and ORB developers must ensure that access to shared data is serialized properly in the ORB and its servants. In addition, the techniques required to robustly terminate servants executing concurrently in multiple threads are complicated, non-portable, and non-intuitive.

• ***Non-extensible*:** The choice of an ORB concurrency strategy depends largely on external factors like application requirements and network/endsystem characteristics. For instance, reactive single-threading [8] is an appropriate strategy for short duration, compute-bound requests on a uni-processor. If these external factors change, however, an ORB's design should be extensible enough to handle alternative concurrency strategies, such as thread pool or thread-per-priority [24].

When ORBs are developed using low-level threading APIs, they are hard to extend with new concurrency strategies *without* affecting other ORB components. For example, adding a thread-per-request architecture to SunSoft IIOP would require extensive changes in order to (1) store the request in a *thread-specific storage* (TSS) variable during protocol processing, (2) pass the key to the TSS variable through the scheduling and demarshaling steps in the Object Adapter, and (3) access the request stored in TSS before dispatching the operation on the servant. Thus, there is no easy way to modify SunSoft IIOP's concurrency architecture without drastically changing its internal structure.

How then can an ORB support a simple, extensible, and portable concurrency mechanism?

Solution $\rightarrow$ the Leader/Followers pattern: An effective way to increase the portability, correctness, and extensibility of ORB concurrency strategies is to apply the *Leader/Follwers* pattern [8]. This pattern provides an efficient concurrency model in which multiple threads take turns to share a set of event sources to detect, demultiplex, dispatch and process service requests that occur on the event sources. In general, the Leader/Followers pattern should be used when an application needs to minimize context switching, synchronization, and data copying, while still allowing multiple threads to run concurrently.

While *Wrapper Facades* provide the basis for portability, they are simply a thin syntactic veneer over the low-level native OS APIs. Moreover, a facade's semantic behavior may still vary across platforms. Therefore, the Leader/Followers pattern defines a higher-level concurrency abstraction that shields TAO from the complexity of low-level thread facades. By raising the level of abstraction for ORB developers, the Leader/Followers pattern makes it easier to define more portable, flexible, and conveniently programmed ORB concurrency strategies. For example, if the number of threads in the pool is 1, the Leader/Followers pattern behaves just like the Reactor pattern.

Using the Leader/Followers pattern in TAO: TAO uses the Leader/Followers pattern to demultiplex GIOP events to `Connection_Handlers` handlers within a pool of threads. When using this pattern, an application pre-spawns a *fixed* number of threads. When these threads invoke TAO's standard `ORB::run` method, one thread will become the leader and wait for a GIOP event. After the leader leader thread detects the event, it promotes an arbitrary thread to become the next leader it and then demultiplexes the event to its associated `Connection_Handler`, which processes the event concurrently with respect to other threads in the ORB. This sequence of steps is shown in Figure 12.

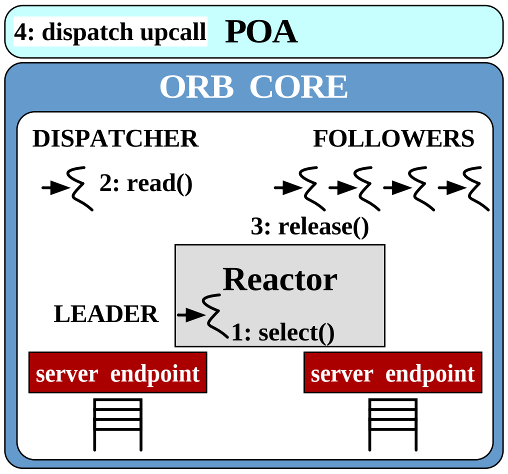


Figure 12: Using the Leader/Followers Pattern to Structure TAO's Concurrency Strategies

As shown in Figure 12, a pool of threads is allocated and a leader thread is chosen to `select` (**1**) on connections for all servants in the server process. When a request arrives, this thread reads (**2**) it into an internal buffer. If this is a valid request for a servant, a follower thread in the pool is released to become the new leader (**3**) and the leader thread dispatches the upcall (**4**). After the upcall is dispatched, the original leader thread becomes a follower and returns to the thread pool. New requests are queued in socket endpoints until a thread in the pool is available to execute the requests.

3.3.5 Reducing Lock Contention and Priority Inversions with the Thread-Specific Storage Pattern

Context: The Leader/Followers pattern allows applications and components in the ORB to run concurrently. The primary drawback to concurrency, however, is the need to *serialize* access to shared resources. In an ORB, common shared resources include the dynamic memory heap, an ORB pseudo-object reference created by the `CORBA::ORB_init` initialization factory, the *Active Object* Map in a POA [22], and the `Acceptor`, `Connector`, and `Reactor` components described earlier. A common way to achieve serialization is to use mutual-exclusion locks on each resource shared by multiple threads.

Problem: In theory, multi-threading an ORB can improve performance by executing multiple instruction streams simultaneously. In addition, multi-threading can simplify internal ORB design by allowing each thread to execute synchronously, rather than reactively or asynchronously. In practice, however, multi-threaded ORBs often perform no better, or even worse, than single-threaded ORBs due to (1) the cost of acquiring/releasing locks and (2) priority inversions that arise when high- and low-priority threads contend for the same locks [40]. In addition, multi-threaded ORBs are hard to program due to complex concurrency control protocols used to avoid race conditions and deadlocks.

Solution → the Thread-Specific Storage pattern: An effective way to minimize the amount of locking required to serialize access to resources shared within an ORB is to use the *Thread-Specific Storage* pattern [8]. This pattern allows multiple threads in an ORB to use one logically global access point to retrieve thread-specific data *without* incurring locking overhead for each access.

In general, the Thread-Specific Storage pattern should be used when the data shared by objects within each thread must be accessed through a globally visible access point that is "logically" shared with other threads, but "physically" unique for each thread.

Using the Thread-Specific Storage Pattern in TAO: TAO uses the Thread-Specific Storage pattern to minimize lock contention and priority inversion for real-time applications. Internally, each thread in TAO uses thread-specific storage to store its ORB Core components, *e.g.*, `Reactor`, `Acceptor`, and `Connector`. When a thread accesses any of these components, they are retrieved by using a `key` as an index into the thread's internal thread-specific state, as shown in Figure 13. Thus, no additional locking is required to access thread-specific ORB state.

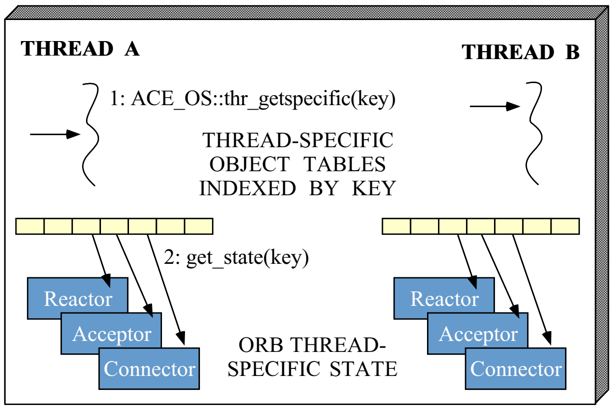

THREAD A
THREAD B
1: ACE_OS::thr_getspecific(key)
THREAD-SPECIFIC OBJECT TABLES INDEXED BY KEY
2: get_state(key)
Reactor
Acceptor
Connector
ORB THREAD-SPECIFIC STATE
Reactor
Acceptor
Connector

Figure 13: Using the Thread-Specific Storage Pattern in TAO

3.3.6 Support Interchangeable ORB Behaviors with the Strategy Pattern

Context: Extensible ORBs must support multiple request demultiplexing and scheduling strategies in their Object Adapters. Likewise, they must support multiple connection establishment, request transfer, and concurrent request processing strategies in their ORB Cores.

Problem: One way to develop an ORB is to provide only static, non-extensible strategies, which are typically configured in the following ways:

• ***Preprocessor macros*:** Some strategies are determined by the value of preprocessor macros. For example, since threading is not available on all OS platforms, conditional compilation is often used to select a feasible concurrency model.

• ***Command-line options*:** Other strategies are controlled by the presence or absence of flags on the command-line. For instance, command-line options can be used to selectively enable ORB concurrency strategies for platforms that support multi-threading [24].

While these two configuration approaches are widely used, they are inflexible. For instance, preprocessor macros only support compile-time strategy selection, whereas command-line options convey a limited amount of information to an ORB. Moreover, these hard-coded configuration strategies are divorced completely from any code they might affect. Thus, ORB components that want to use these options must (1) know of their existence, (2) understand their range of values, and (3) provide an appropriate implementation for each value. Such restrictions make it hard to develop highly extensible ORBs that are composed from transparently configurable strategies.

How then does an ORB (1) permit replacement of subsets of component strategies in a manner orthogonal and transparent to other ORB components and (2) encapsulate the state and behavior of each strategy so that changes to one component do not permeate throughout an ORB haphazardly?

Solution → the Strategy pattern: An effective way to support multiple transparently "pluggable" ORB strategies is to apply the *Strategy* pattern [15]. This pattern factors out similarities among algorithmic alternatives and explicitly associates the name of a strategy with its algorithm and state. Moreover, the Strategy pattern removes lexical dependencies on strategy implementations since applications access specialized behaviors only through common base class interfaces. In general, the Strategy pattern should be used when an application's behavior can be configured via multiple interchangeable strategies.

Using the Strategy Pattern in TAO: TAO uses a variety of strategies to factor out behaviors that are often hard-coded in conventional ORBs. Several of these strategies are illustrated in Figure 14. For instance, TAO supports multiple re-

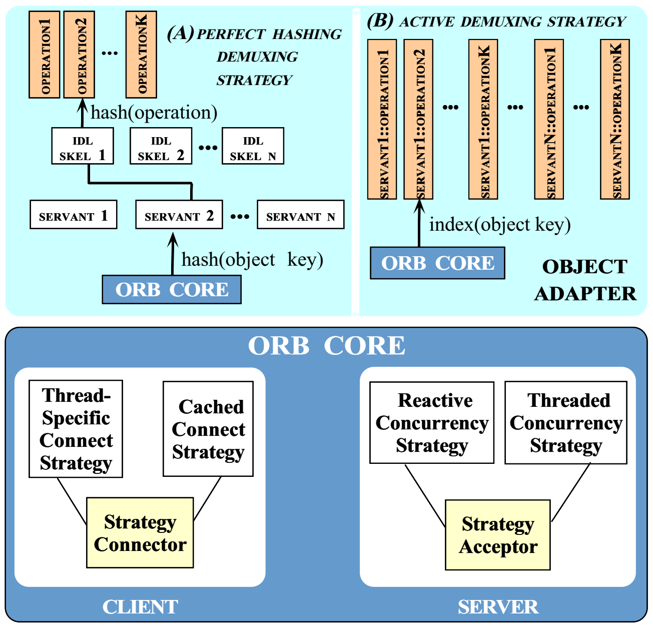


Figure 14: ORB Core and POA Strategies in TAO

quest demultiplexing strategies (*e.g.*, perfect hashing vs. active demultiplexing [36]) and dispatching strategies (*i.e.*, FIFO vs. rate-based) in its Object Adapter, as well as connection management strategies (*e.g.*, process-wide cached connections vs. thread-specific cached connections) and handler concurrency strategies (*e.g.*, Reactive vs. variations of Leader/Followers) in its ORB Core.

3.3.7 Consolidate ORB Strategies Using the Abstract Factory Pattern

Context: There are many potential strategy variants supported by TAO. Table 1 shows a simple example of the strategies used to create two configurations of TAO. Configuration 1

	Strategy Configuration			
Application	*Concurrency*	*Dispatching*	*Demultiplexing*	*Protocol*
Avionics	Thread-per priority	Priority -based	Perfect hashing	VME backplane
Medical Imaging	Thread-per connection	FIFO	Active demultiplexing	TCP/IP

Table 1: Example Applications and their ORB Strategy Configurations

is an avionics application with deterministic real-time requirements [11]. Configuration 2 is an electronic medical imaging application [41] with high throughput requirements. In general, the forces that must be resolved to compose all ORB strategies correctly are the need to (1) ensure the configuration of semantically compatible strategies and (2) simplify the management of a large number of individual strategies.

Problem: An undesirable side-effect of using the Strategy pattern extensively in complex ORB software–as well as other types of software–is that it becomes hard to manage extensibility for the following reasons:

• ***Complicated configuration and evolution*:** ORB source code can become littered with hard-coded references to strategy types, which complicates configuration and evolution. For example, within a particular application domain, such as real-time avionics or medical imaging, many independent strategies must act harmoniously. Identifying these strategies individually by name, however, requires tedious replacement of selected strategies in one domain with a potentially different set of strategies in another domain.

• ***Semantic incompatibilities*:** It is not always possible for certain ORB strategy configurations to interact compatibly. For instance, the FIFO strategy for scheduling requests shown in Table 1 may not work with the thread-per-priority concurrency architecture. The problem stems from semantic incompatibilities between scheduling requests in their order of arrival (*i.e.*, FIFO queueing) vs. dispatching requests based on their relative priorities (*i.e.*, preemptive priority-based thread dispatching). Moreover, some strategies are only useful when certain preconditions are met. For instance, the perfect hashing demultiplexing strategy is generally feasible only for systems that statically configure all servants off-line [22].

How can a highly-configurable ORB reduce the complexities required to manage its myriad strategies, as well as enforce semantic consistency when combining discrete strategies?

Solution → the Abstract Factory pattern: An effective way to consolidate multiple ORB strategies into semantically compatible configurations is to apply the *Abstract Factory* pattern [15]. This pattern provides a single access point that integrates all strategies used to configure an ORB. Concrete subclasses then aggregate compatible application-specific or domain-specific strategies, which can be replaced *en masse* in semantically meaningful ways. In general, the Abstract Factory pattern should be used when an application must consolidate the configuration of many strategies, each having multiple alternatives that must vary together.

Using the Abstract Factory pattern in TAO: All of TAO's ORB strategies are consolidated into two abstract factories that are implemented as Singletons [15]. One factory encapsulates client-specific strategies, the other factory encapsulates server-specific strategies, as shown in Figure 15. These abstract factories encapsulate request demultiplexing, scheduling, and dispatch strategies in the server, as well as concurrency strategies in both client and server. By using the Abstract Factory pattern, TAO can configure different ORB personalities conveniently and consistently.

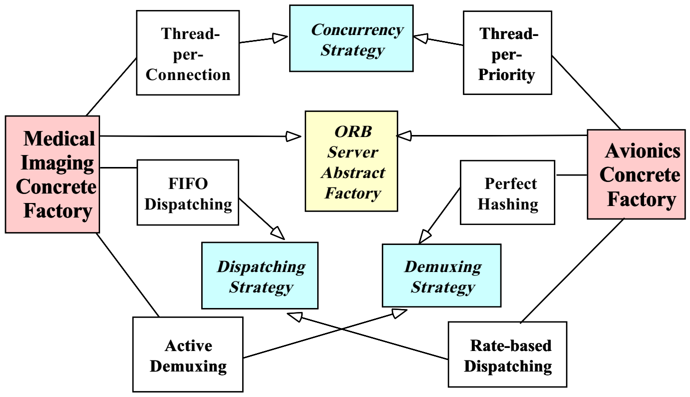


Figure 15: Factories used in TAO

3.3.8 Dynamically Configure ORBs with the Component Configurator Pattern

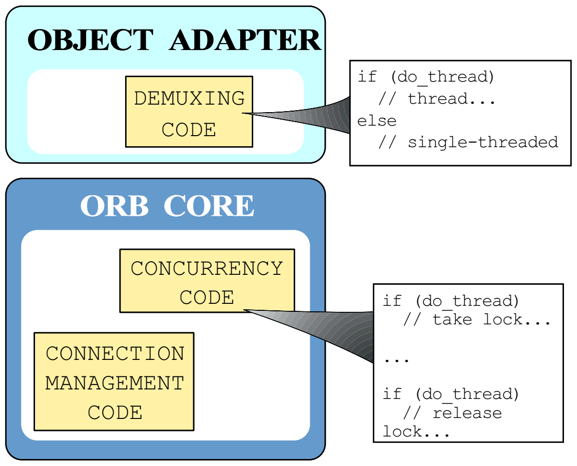
Context: The cost of many computing resources, such as memory and CPUs, continues to drop. However, ORBs must still avoid excessive consumption of finite system resources. This parsimony is particularly essential for embedded and real-time systems that require small memory footprints and predictable CPU utilization [20]. Many applications can also benefit from the ability to extend ORBs *dynamically*, *i.e.*, by allowing their strategies to be configured at run-time.

Problem: Although the Strategy and Abstract Factory patterns simplify the customization of ORBs for specific application requirements and system characteristics, these patterns can still cause the following problems for extensible ORBs:

• ***High resource utilization***: Widespread use of the Strategy pattern can substantially enlarge the number of strategies configured into an ORB, which can increase the system resources required to run an ORB.

• ***Unavoidable system downtime***: If strategies are configured statically at compile-time or static link-time using abstract factories, it is hard to enhance existing strategies or add new strategies without (1) changing the existing source code for the consumer of the *strategy* or the *abstract factory*, (2) recompiling and relinking an ORB, and (3) restarting running ORBs and their application servants.

Although it does not use the Strategy pattern explicitly, SunSoft IIOP does permit applications to vary certain ORB strategies at run-time. However, these different strategies must be configured statically into SunSoft IIOP at compile-time. Moreover, as the number of alternatives increases, so does the amount of code required to implement them. For instance, Figure 16 illustrates SunSoft IIOP's approach to varying the concurrency *strategy*.



```
if (do_thread)
  // thread...
else
  // single-threaded
```

```
if (do_thread)
  // take lock...
...
if (do_thread)
  // release
lock...
```

Figure 16: SunSoft IIOP Hard-coded Strategy Usage

Each area of code that might be affected by the choice of concurrency strategy is trusted to act independently of other areas. This proliferation of decision points adversely increases the complexity of the code, complicating future enhancement and maintenance. Moreover, the selection of the data type specifying the strategy complicates integration of new concurrency architectures because the type (`bool`) would have to change, as well as the programmatic structure, `if (do_thread) then ... else ...`, that decodes the strategy specifier into actions.

In general, static configuration is only feasible for a small, fixed number of strategies. However, configuring complex

ORB middleware (1) statically complicates evolution, (2) increases system resource utilization, and (3) leads to unavoidable system downtime to modify existing components.

How then does an ORB implementation reduce the "overly-large, overly-static" side-effects stemming from pervasive use of the Strategy and Abstract Factory patterns?

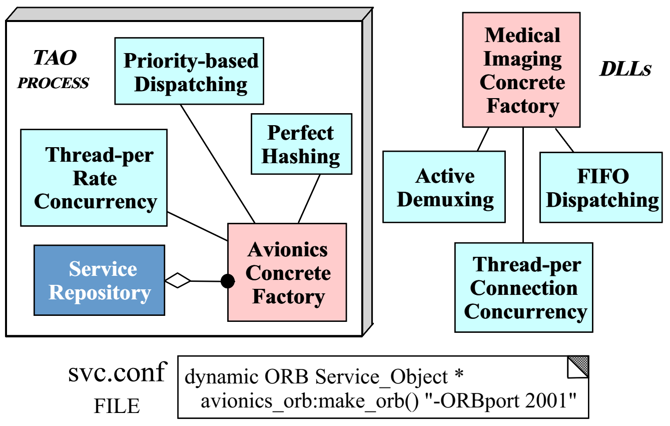
Solution $\rightarrow$ the Component Configurator pattern: An effective way to enhance the dynamism of an ORB is to apply the *Component Configurator* pattern [8]. This pattern uses explicit dynamic linking [28] mechanisms to obtain, utilize, and/or remove the run-time address bindings of custom strategy and abstract factory objects into an ORB at installation-time and/or run-time. Widely available explicit dynamic linking mechanisms include the `dlopen/dlsym/dlclose` functions in SVR4 UNIX [42] and the `LoadLibrary/GetProcAddress` functions in the WIN32 subsystem of Windows NT [43]. The ACE wrapper facades used by TAO portably encapsulate these OS APIs.

By using the Component Configurator pattern, the *behaviors* of ORB strategies are decoupled from *when* the strategy implementations are configured into an ORB. For instance, ORB strategies can be linked into an ORB from dynamically linked libraries (DLL)s at compile-time, installation-time, or even during run-time. Moreover, the Component Configurator pattern can reduce the memory footprint of an ORB by allowing application developers and/or system administrators to dynamically link only those strategies that are necessary for a specific ORB personality.

In general, the Component Configurator pattern should be used when (1) an application wants to configure its constituent components dynamically and (2) conventional techniques, such as command-line options, are insufficient due to the number of possibilities or the inability to anticipate the range of values.

Using the Component Configurator pattern in TAO: TAO uses the Component Configurator pattern in conjunction with the Strategy and Abstract Factory patterns to dynamically install the strategies it requires without (1) recompiling or statically relinking existing code or (2) terminating and restarting an existing ORB and its application servants. This design allows the behavior of TAO to be tailored for specific platforms and application requirements without requiring access to, or modification of, ORB source code.

In addition, the Component Configurator pattern allows applications to customize the personality of TAO at run-time. For instance, during TAO's ORB initialization phase, it uses the dynamic linking mechanisms provided by the OS (and encapsulated by the ACE wrapper facades) to link in the appropriate concrete factory for a particular use-case. Figure 17 shows two factories tuned for different application domains supported by TAO: avionics and medical imaging.


TAO
PROCESS
Priority-based Dispatching
Perfect Hashing
Thread-per Rate Concurrency
Avionics Concrete Factory
Service Repository
Medical Imaging Concrete Factory
DLLs
Active Demuxing
FIFO Dispatching
Thread-per Connection Concurrency
svc.conf
FILE

```
dynamic ORB Service_Object *
  avionics_orb:make_orb() "-ORBport 2001"
```

Figure 17: Using the Component Configurator Pattern in TAO

In the configuration shown in Figure 17, the Component Configurator has consulted the `comp.conf` script and installed the avionics concrete factory in the process. Applications using this ORB personality will be configured with a particular set of ORB concurrency, demultiplexing, and dispatching strategies. The medical imaging concrete factory resides in a DLL outside of the existing ORB process. To configure a different ORB personality, this factory could be installed dynamically during TAO's ORB server initialization phase.

3.4 Summary of Design Challenges and Patterns That Resolve Them

Table 2 summarizes the mapping between ORB design challenges and the patterns in the pattern language that we applied to resolve these challenges in TAO. This table focuses on the forces resolved by individual patterns. However, TAO also benefits from the collaborations among *multiple* patterns in the pattern language. For example, the Acceptor and Connector patterns utilize the Reactor pattern to notify them when connection events occur at the OS level.

Forces	Resolving Pattern
Abstracting low-level system calls	*Wrapper Facade*
ORB event demultiplexing	*Reactor*
ORB connection management	*Acceptor-Connector*
Efficient concurrency models	*Leader/Followers*
Pluggable strategies	*Strategy*
Group similar initializations	*Abstract Factory*
Dynamic run-time configuration	*Component Configurator*

Table 2: Summary of Forces and Their Resolving Patterns

Moreover, patterns often must collaborate to alleviate drawbacks that arise from applying them in isolation. For instance, the reason the Abstract Factory pattern is used in TAO is to avoid the complexity caused by its extensive use of the Strategy pattern. Although the Strategy pattern simplifies the effort required to customize an ORB for specific application requirements and network/endsystem characteristics, it is tedious and error-prone to manage a large number of strategy interactions manually.

3.5 Evaluating the Contribution of Patterns to ORB Middleware

Section 3.3 described the pattern language used in TAO and qualitatively evaluated how these patterns helped to alleviate limitations with the design of SunSoft IIOP. The discussion below goes one step further and quantitatively evaluates the benefits of applying patterns to ORB middleware.

3.5.1 Where's the Proof?

Implementing TAO using a pattern language yielded significant quantifiable improvements in software reusability and maintainability. The results are summarized in Table 3. This table compares the following metrics for TAO and SunSoft IIOP:

1. The number of methods required to implement key ORB tasks (such as connection management, request transfer, socket and request demultiplexing, marshaling, and dispatching).
2. The total non-comment lines of code (LOC) for these methods.
3. The average McCabe Cyclometric Complexity metric $v(G)$ [44] of the methods. The $v(G)$ metric uses graph theory to correlate code complexity with the number of possible basic paths that can be taken through a code module. In C++, a module is defined as a method.

The use of patterns in TAO significantly reduced the amount of *ad hoc* code and the complexity of certain operations. For instance, the total lines of code in the client-side *Connection Management* operations were reduced by a factor of 5. Moreover, the complexity for this component was substantially reduced by a factor of 16. These reductions in LOC and complexity stem from the following factors:

- These ORB tasks were the focus of our initial work when developing TAO.
- Many of the details of connection management and socket demultiplexing were subsumed by patterns and components in the ACE framework, in particular, the Acceptor, Connector, and Reactor.

Other areas did not yield as much improvement. In particular, *GIOP Invocation* tasks actually increased in size and maintained a consistent $v(G)$. There were two reasons for this increase:

1. The primary pattern applied in these cases was the Wrapper Facade, which replaced the low-level system calls with ACE wrappers but did not factor out common strategies; and
2. SunSoft IIOP did not trap all the error conditions, which TAO addressed much more completely. Therefore, the additional code in TAO is necessary to provide a more robust ORB.

The most compelling evidence that the systematic application of patterns can positively contribute to the maintainability of complex software is shown in Figure 18. This figure illustrates the distribution of $v(G)$ over the percentage of affected methods in TAO. As shown in the figure, most of TAO's code is structured in a straightforward manner, with almost 70% of the methods' $v(G)$ falling into the range of 1-5.

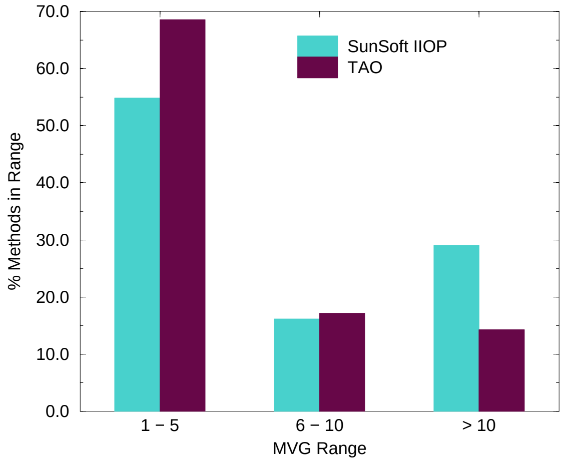


Figure 18: Distribution of $v(G)$ Over ORB Methods

In contrast, while SunSoft IIOP has a substantial percentage (55%) of its methods in that range, many of the remaining methods (29%) have $v(G)$ greater than 10. The reason for the difference is that SunSoft IIOP uses a monolithic coding style with long methods. For example, the average length of methods with $v(G)$ over 10 is over 80 LOC. This yields overly-complex code that is hard to debug and understand.

In TAO, most of the monolithic SunSoft IIOP methods were decomposed into smaller methods when integrating the patterns. The majority (86%) of TAO's methods have $v(G)$ under 10. Of that number, nearly 70% have a $v(G)$ between 1 and 5.

	TAO			SunSoft IIOP		
ORB Task	*# Methods*	*Total LOC*	*Avg.* $v(G)$	*# Methods*	*Total LOC*	*Avg.* $v(G)$
Connection Management (Server)	2	43	7	3	190	14
Connection Management (client)	3	11	1	1	64	16
GIOP Message Send (client/Server)	1	46	12	1	43	12
GIOP Message Read (client/Server)	1	67	19	1	56	18
GIOP Invocation (client)	2	205	26	2	188	27
GIOP Message Processing (client/Server)	3	41	2	1	151	24
Object Adapter Message Dispatch (Server)	2	79	6	1	61	10

Table 3: Code Statistics: TAO vs. SunSoft IIOP

The relatively few (14%) methods in TAO with $v(G)$ greater than 10 are largely unchanged from the original SunSoft IIOP TypeCode interpreter. Subsequent releases of TAO have completely removed the TypeCode interpreter and replaced it with stubs and skeletons generated automatically by TAO's IDL compiler. Thus, there is no need for TAO ORB developers to maintain this code anymore.

In general, the use of monolithic methods in SunSoft IIOP not only increased its maintenance effort, it also degrades its performance due to reduced processor cache hits [20]. Therefore, we plan to experiment with the application of other patterns, such as *Command* and *Template Method* [15], to simplify and optimize these monolithic methods into smaller, more cohesive methods.

3.5.2 What are the Benefits?

In general, the applying a pattern language to TAO yielded the following benefits:

Increased extensibility: Patterns such as Abstract Factory, Strategy, and Component Configurator simplify the configuration of TAO for a particular application domain by allowing extensibility to be "designed into" the ORB. In contrast, DOC middleware lacking these patterns is significantly harder to extend.

Enhanced design clarity: By applying a pattern language to TAO, not only did we develop a more extensible ORB, we also devised a richer vocabulary for expressing ORB middleware designs. In particular, a pattern language captures and articulates the design rationale for complex object-structures in an ORB. Moreover, it helps to demystify and motivate the structure of an ORB by describing its architecture in terms of design forces that recur in many types of software systems. The expressive power of a pattern language enabled us to concisely convey the design of complex software systems, such as TAO. As we continue to learn about ORBs and the patterns of which they are composed, we expect our pattern vocabulary to grow and evolve into an even more comprehensive pattern language.

Increased portability and reuse: TAO is built atop the ACE framework, which provides implementations of many key communication software patterns[28]. Using ACE simplified the porting of TAO to numerous OS platforms since most of the porting effort was absorbed by the ACE framework maintainers. In addition, since the ACE framework is rich with configurable high-performance, real-time network-oriented components, we were able to achieve considerable code reuse by leveraging the framework. This is indicated by the consistent decrease in lines of code (LOC) in Table 3.

3.5.3 What are the Liabilities?

The use of a pattern language can also incur some liabilities. We summarize these liabilities below and discuss how we minimize them in TAO.

Abstraction penalty: Many patterns use indirection to increase component decoupling. For instance, the Reactor pattern uses virtual methods to separate the application-specific `Event Handler` logic from the general-purpose event demultiplexing and dispatching logic. The extra indirection introduced by using these pattern implementations can potentially decrease performance. To alleviate these liabilities, we carefully applied C++ programming language features (such as inline functions and templates) and other optimizations (such as eliminating demarshaling overhead [20] and demultiplexing overhead [36]) to minimize performance overhead. As a result, TAO is substantially faster than the original hard-coded SunSoft IIOP [20].

Additional external dependencies: Whereas SunSoft IIOP only depends on system-level interfaces and libraries, TAO depends on wrapper facades in the ACE framework. Since ACE encapsulates a wide range of low-level OS mechanisms, the effort required to port it to a new platform could potentially be higher than porting SunSoft IIOP, which only uses a subset of the OS's APIs. However, since ACE has been ported to many platforms already, the effort to port to new platforms is relatively low. Most sources of platform variation have been isolated to a few modules in ACE.

4 Concluding Remarks

This paper presented a case study illustrating how we applied a pattern language to enhance the extensibility of TAO, which is a dynamically configurable ORB that is targeted for distributed applications with high-performance and real-time requirements. We found qualitative and quantitative evidence that the use of this pattern language helped to clarify the structure of, and collaboration between, components that perform key ORB tasks. These tasks include event demultiplexing and event handler dispatching, connection establishment and initialization of application services, concurrency control, and dynamic configuration. In addition, patterns improved TAO's performance and predictability by making it possible to transparently configure lightweight and optimized strategies for processing client requests.

A principal benefit of applying a pattern language to guide TAO's design is that the systematic application of patterns in the language improved the decoupling and object-oriented structure of the ORB significantly. The patterns we used were applied in roughly the same order that they appear in Section 3.3. Each evolution of TAO leveraged upon the results of prior evolutions. This iterative process revealed new insights on which patterns in the language could be applied and how they might be applied in subsequent stages.

The complete C++ source code, examples, and documentation for ACE and TAO is freely available at URL `www.cs.wustl.edu/~schmidt/TAO.html`.

References

[1] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

[2] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[3] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[5] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[6] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[7] D. C. Schmidt, "Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software," *Communications of the ACM (Special Issue on Object-Oriented Experiences)*, vol. 38, October 1995.

[8] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*. New York, NY: Wiley & Sons, 2000.

[9] D. C. Schmidt, "Applying a Pattern Language to Develop Application-level Gateways," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[10] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[11] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[12] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

[13] C. O'Ryan, D. C. Schmidt, and D. Levine, "Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations," in *Proceedings of the 5^{th} Workshop on Object-oriented Real-time Dependable Systems*, (Montery, CA), IEEE, Nov. 1999.

[14] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[16] C. O'Ryan, F. Kuhns, D. C. Schmidt, and J. Parsons, "Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[17] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

[18] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[19] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[20] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[21] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2^{nd} C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[22] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazine*, vol. 8, no. 1, 2000.

[23] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, to appear 2000.

[24] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2000.

[25] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[26] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, 2000.

[27] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[28] D. C. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[29] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[30] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0," in *Proceedings of the* 6^{th} *IEEE Real-Time Technology and Applications Symposium*, (Washington DC), IEEE, May 2000.

[31] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the* 2^{nd} *International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Sept. 2000.

[32] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "Applying Patterns to Improve the Performance of Fault-Tolerant CORBA," in *Proceedings of the* 7^{th} *International Conference on High Performance Computing (HiPC 2000)*, (Bangalore, India), ACM/IEEE, Dec. 2000.

[33] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[34] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[35] M. Roman, M. D. Mickunas, F. Kon, and R. Campbell, "LegORB and Ubiquitous CORBA," in *Reflective Middleware Workshop*, ACM/IFIP, Apr. 2000.

[36] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[37] J. Hu and D. C. Schmidt, "JAWS: A Framework for High Performance Web Servers," in *Domain-Specific Application Frameworks: Frameworks Experience by Industry* (M. Fayad and R. Johnson, eds.), Wiley & Sons, 1999.

[38] Object Management Group, *OMG Real-time Request for Proposal*, OMG Document ptc/97-06-20 ed., June 1997.

[39] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[40] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[41] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[42] R. Gingell, M. Lee, X. Dang, and M. Weeks, "Shared Libraries in SunOS," in *Proceedings of the Summer 1987 USENIX Technical Conference*, (Phoenix, Arizona), 1987.

[43] D. A. Solomon, *Inside Windows NT, 2nd Ed.* Redmond, Washington: Microsoft Press, 2nd ed., 1998.

[44] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, Dec. 1976.