

Applying Patterns to Develop Extensible and Maintainable ORB Middleware

Id : ORB – patterns.tex, v1.831998/01/1419 : 17 : 28levineExp

Douglas C. Schmidt and Chris Cleeland

{schmidt,cleeland}@cs.wustl.edu

Department of Computer Science

Washington University

St. Louis, MO 63130, USA

(314) 935-7538*

Abstract

Distributed object computing forms the basis for next-generation application middleware. At the heart of distributed object computing are Object Request Brokers (ORBs), which automate many tedious and error-prone distributed programming tasks. Like many other distributed applications, conventional ORBs use statically configured software designs, which are hard to maintain, port, and optimize. 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 article makes two contributions to the study of extensible and maintainable ORB middleware. First, it presents a case study of key design patterns needed to develop ORBs that can be dynamically configured and evolved for specific application requirements and system characteristics. Second, we quantify the impact of applying patterns to reduce the complexity and improve the maintainability of common ORB tasks.

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] such as ActiveX and Java Beans. Second, there is great demand for *distribution technology* such as remote method invocation or message-oriented middleware that simplifies application collaboration within and across enterprises [2]. Third, there are increasing efforts to define standard software infrastructures such as CORBA that allow applications to interwork seamlessly throughout *hetero-*

geneous environments [3]. Finally, next-generation distributed applications such as video-on-demand and teleconferencing require Quality of Service (QoS) guarantees for latency, bandwidth, and reliability [4].

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 maintaining distributed applications. In particular, DOC middleware automates common network programming tasks such as object location, object activation, parameter marshaling, fault recovery, and security. At the heart of DOC middleware are *Object Request Brokers (ORBs)*, such as CORBA [5], DCOM [6], and Java RMI [7].

This article describes how *design patterns* can be used to develop *dynamically configurable* ORB middleware that can be extended and maintained more easily than conventional *statically configured* middleware. Generally defined, a pattern represents a recurring solution to a software development problem within a particular context [8]. Patterns help to alleviate the continual re-discovery and re-invention of software concepts and components by documenting proven solutions to standard software development problems [9]. For instance, patterns are useful for documenting the structure and participants in common communication software micro-architectures like Reactors [10], Active Objects [11], and Brokers [12]. These patterns are generalizations of object-structures that have been used successfully to build flexible and efficient event-driven and concurrent communication software such as ORBs.

To focus the discussion, this article presents a case study that illustrates how we have applied patterns to develop *The ACE ORB (TAO)* [13]. TAO is a freely available, highly extensible ORB targeted for applications with real-time quality of

*This research is supported in part by grants from Boeing, NSF grant NCR-9628218, Siemens, and Sprint.

service (QoS) requirements. These applications include avionics mission computers [14], telecommunication switch management systems [10], and electronic medical imaging systems [15]. A novel aspect of TAO is its extensible design, which can be customized dynamically to meet specific application QoS requirements and network/endsystem characteristics.

The remainder of this article is organized as follows: Section 2 gives an overview of CORBA and TAO; Section 3 motivates the need for dynamic configuration and describes the patterns that resolve key design challenges faced when developing and maintaining extensible ORBs; Section 4 evaluates and quantifies the contribution of patterns to ORB middleware; and Section 5 presents concluding remarks.

2 Overview of CORBA and TAO

2.1 Overview of the CORBA ORB Reference Model

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for:

- **Object location:** CORBA objects can be located locally with the client or remotely on a server, without affecting their 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 like VxWorks, Chorus, and LynxOS.
- **Communication protocols and interconnects:** The communication protocols and interconnects that CORBA can run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, and embedded system backplanes.
- **Hardware:** CORBA shields applications from differences in hardware such as RISC vs. CISC instruction sets.

Figure ?? illustrates the following components, all of which are composed to provide the transparency above:

Servant: This component implements the operations defined by an OMG Interface Definition Language (IDL) interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are implemented using one or more objects. A servant is identified by its *object reference*, which uniquely identifies the servant in a server process.

Client: This program entity performs application tasks by obtaining object references to servants and invoking operations on the servants. Servants can be remote or co-located relative to the client. Ideally, accessing a remote servant should be as simple as calling an operation on a local object, *i.e.*, `object->operation(args)`. Figure ?? shows the components that ORBs use to transmit requests transparently from client to servant for remote operation invocations.

ORB Core: When a client invokes an operation on a servant, the ORB Core is responsible for delivering the request to the servant and returning a response, if any, to the client. For servants executing remotely, a CORBA-compliant [5] ORB Core communicates via the Internet Inter-ORB Protocol (IIOP), a version of the General Inter-ORB Protocol (GIOP) which runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked into client and server applications.

ORB Interface: An ORB is a logical entity that may be implemented in various ways, *e.g.*, one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This ORB interface provides standard operations that convert object references to strings and back, creates argument lists for requests made through the Dynamic Invocation Interface (DII) described below, as well as others.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a “glue” between the client and servants, respectively, and the ORB. Stubs provide a strongly-typed, *static* invocation interface (SII) that marshals application data into a common packet-level representation. Conversely, skeletons demarshal the packet-level representation back into typed data that is meaningful to an application. An IDL compiler automatically transforms OMG IDL definitions into an application programming language like C++ or Java. IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [16].

Dynamic Invocation Interface (DII): The DII allows a client to access the underlying request transport mechanisms provided by the ORB Core. The DII is useful when an application has no compile-time knowledge of the interface it is accessing. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of twoway operations to avoid blocking the client until the servant responds. In contrast, SII stubs support only twoway (*i.e.*, request/response) and oneway (*i.e.*, request only) operations.¹

¹The OMG is currently standardizing an asynchronous method invocation interface, as well.

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 a servant that has no compile-time knowledge of the IDL interface it is implementing. Clients making requests need not know whether the server ORB uses static skeletons or dynamic skeletons.

Object Adapter: An Object Adapter associates a servant with an ORB, demultiplexes incoming requests to the servant, and dispatches the appropriate operation upcall on that servant. Recent CORBA portability enhancements [?] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB.

2.2 Overview of TAO

The TAO project focuses on the following topics related to real-time CORBA and ORB endsystems:

- Identifying enhancements to standard ORB specifications, particularly OMG CORBA, that will enable applications to specify their QoS requirements concisely to ORB endsystems [17].
- Empirically determining the features required to build real-time ORB endsystems that can enforce deterministic and statistical end-to-end application QoS guarantees [13].
- Integrating the strategies for I/O subsystem architectures and optimizations [18] with ORB middleware to provide end-to-end bandwidth, latency, and reliability guarantees to distributed applications.
- Capturing and documenting the key design patterns [19] necessary to develop, maintain, configure, and extend real-time ORB endsystems.

The focus of this article is on design patterns for developing extensible ORB middleware.

TAO is an ORB endsystem that contains the network interface, operating system, communication protocol, and CORBA middleware components and features shown in Figure 1. TAO is based on the standard OMG CORBA reference model, with the following enhancements designed to overcome the shortcomings of conventional ORBs for high-performance and real-time applications:

- **Real-time IDL Stubs and Skeletons:** In addition to marshaling and demarshaling of operation parameters, TAO’s Real-time IDL (RIDL) stubs and skeletons are responsible for ensuring that application timing requirements are specified and enforced end-to-end [20].

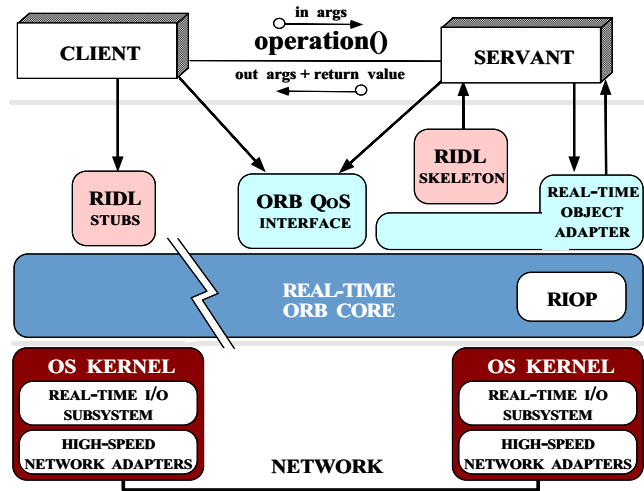


Figure 1: Components in the TAO Real-time ORB

- **Real-time Object Adapter and ORB Core:** In addition to associating servants with the ORB and demultiplexing incoming requests to servants, TAO’s Object Adapter (OA) implementation dispatches servant operations in accordance with various real-time scheduling strategies such as Rate Monotonic and Earliest Deadline First [13].
- **ORB QoS Interface:** Applications can use TAO’s QoS interface to map real-time processing requirements to ORB endsystem/network resources. Common real-time processing requirements include end-to-end latency bounds and periodic scheduling deadlines. Common ORB endsystem/network resources include CPU, memory, network connections and storage devices.
- **Real-time I/O subsystem:** TAO’s real-time I/O subsystem performs admission control and assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be guaranteed.
- **High-speed network adapters:** TAO’s I/O subsystem contains a “daisy-chained” interconnect comprising a number of ATM Port Interconnect Controller (APIC) chips [21]. APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps. However, TAO also runs on conventional real-time interconnects such as VME backplanes and multi-processor shared memory environments.

To expedite our project goals, and to avoid re-inventing existing components, we based TAO on SunSoft IIOIP, which is a freely available² reference implementation of the Internet Inter-ORB Protocol (IIOIP). SunSoft IIOIP is written in C++

²See <ftp://ftp.omg.org/pub/interop/> for the source code.

and provides many features of a CORBA ORB. However, SunSoft IIOP has the following limitations:

- **Lack of key ORB features:** Although SunSoft IIOP provides an ORB Core, a robust IIOP protocol engine, and a DII and DSI implementation, it lacks an IDL compiler, an Interface Repository, and a Portable Object Adapter (POA). TAO implements many of these missing features and also provides several new features such as real-time scheduling and dispatching mechanisms [13].

- **Lack of portability:** Like most communication software, SunSoft IIOP is directly programmed with low-level networking and OS APIs such as sockets, `select`, and POSIX Pthreads. In addition to being tedious and error-prone, these APIs are not portable across OS platforms; many operating systems like Win32 and VxWorks, lack features like Pthreads. Section 3.3.1 illustrates how we used the Wrapper Facade pattern [8] to improve TAO's portability.

- **Lack of configurability:** Like many ORBs and other middleware, SunSoft IIOP is *statically* configured, 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 [8], Strategy [8], and Service Configurator [22] patterns to simplify the configurability of TAO 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 accomplishes 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 coupling and increase cohesion in TAO.

3 Using Patterns to Build an Extensible ORB Middleware

3.1 Why We Need Dynamically Configurable Middleware

A key motivation for ORB middleware is to offload complex distributed system infrastructure tasks from application developers to ORB developers. In this model, ORB developers are responsible for implementing reusable middleware components that handle common tasks such as interprocess communication, concurrency, endpoint demultiplexing, scheduling, and dispatching. 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 conventional ORBs can only be 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 [23, 22]:

- **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, which 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 like telecom call processing that require 7×24 availability.

- **Inefficiency:** Statically-configured ORBs can be inefficient, both in terms of space and time. Space inefficiency can stem 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.

Time inefficiency can stem from restricting an ORB to use statically configured algorithms or data structures for key processing tasks. This can make it hard for application developers to customize an ORB to handle new user-cases. For instance, real-time avionics mission computing systems [14] can instantiate all their servants off-line. These systems can benefit from an ORB that uses perfect hashing [24] to demultiplex incoming requests to servants. However, ORBs that are configured statically to use a general-purpose, "one-size-fits-all" demultiplex strategy will not perform as well for mission computing 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 and maintainability is to develop ORB middleware that can be *dynamically configured*.

Dynamic configuration enables the selective integration of customized implementations for key ORB strategies such as communication, concurrency, demultiplexing, scheduling, and dispatching. This 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 make these decisions very late in the design lifecycle, *i.e.*, at installation-time or run-time.

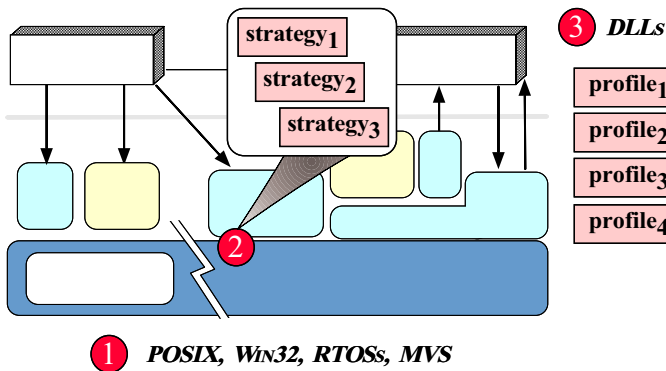


Figure 2: Dimensions of ORB Extensibility

Figure 2 illustrates the following key dimensions of ORB extensibility:

1. Extensibility to retargeting 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 like 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 instance, ORB components can be customized to meet periodic deadlines in real-time systems. 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 dynamically from 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.

This section describes patterns that we have applied to enhance the extensibility of TAO along each dimension outlined above.

3.2 Patterns for Improving ORB Extensibility

This section uses TAO as a case study to illustrate how patterns can help application developers and ORB developers build, maintain, and extend communication software by reducing the coupling between components. The patterns described in this section are not limited to ORBs or communication middleware, however. 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.

Figure 3 illustrates the patterns used to develop an extensible ORB architecture for TAO. It is beyond the scope of this

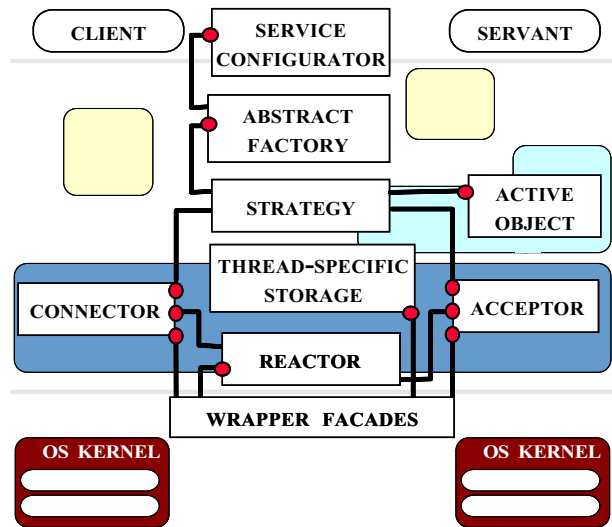


Figure 3: Relationships Among Patterns Used in TAO

section to describe each pattern in detail or to discuss all the patterns used within TAO. Instead, our goal is to focus on the key patterns and show how they can improve the extensibility, maintainability, and performance of complex ORB middleware. The references point to additional material on each pattern.

The intent and usage of these patterns are outlined below:

The Wrapper Facade pattern: which simplifies the OS system programming interface by combining multiple related OS system calls like the socket API or POSIX Pthreads into cohesive OO abstractions [8]. TAO uses this pattern to avoid tedious, non-portable, and non-type-safe programming of low-level, OS-specific system calls.

The Reactor pattern: which provides flexible event demultiplexing and event handler dispatching [10]. 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.

The Acceptor-Connector pattern: which decouples GIOP protocol handler initialization from the ORB processing tasks performed once initialization is complete [25]. 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.

The Active Object pattern: which supports flexible concurrency architectures by decoupling request reception from request execution [11]. TAO uses this pattern to facilitate the use of multiple concurrency strategies that can be configured flexibly into its ORB Core at run-time.

The Thread-Specific Storage pattern: which allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access [26]. TAO uses this pattern to minimize lock contention and priority inversion for real-time applications.

The Strategy pattern: which provides an abstraction for selecting one of several candidate algorithms and packaging it into an object [8]. This pattern is the foundation of TAO’s extensible software architecture and makes it possible to configure custom ORB strategies for concurrency, communication, scheduling, and demultiplexing.

The Abstract Factory pattern: which provides a single factory 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 explicitly specifying their subclass name [8].

The Service Configurator pattern: which permits dynamic run-time configuration of abstract factories and strategies in an ORB [22]. TAO uses this pattern to dynamically interchange abstract factory implementations in order to customize ORB personalities at run-time.

3.3 How to Resolve ORB Design Challenges with Patterns

In the following, we outline the forces that underlie the main design challenges related to developing extensible and maintainable ORBs. We also explain how the absence of these patterns in conventional ORBs like SunSoft IIOp leaves these

forces unresolved. In addition, we describe which patterns resolve these forces and illustrate how TAO implements these patterns to create an extensible and maintainable real-time ORB.

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

Context: One role of an ORB is to shield application-specific servants from the details of low-level systems programming. Therefore, ORB developers, rather than application developers, are responsible for tedious, low-level tasks like demultiplexing events, sending and receiving requests from the network, and spawning threads to execute requests concurrently. Figure 4 illustrates a common approach used by SunSoft IIOp, which is programmed internally using system APIs like sockets, `select`, and POSIX Pthreads directly.

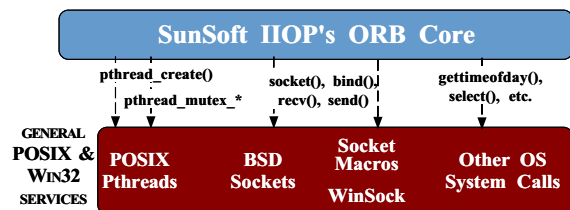


Figure 4: SunSoft IIOp Operating System Interaction

Problem: Developing an ORB is difficult. It is even more difficult if developers must wrestle with low-level system APIs written in languages like C, which often causes the following problems:

- **ORB developers must have intimate knowledge of many OS platforms:** Implementing an ORB using system-level APIs forces developers to deal with the non-portable, tedious, and error-prone OS idiosyncrasies such as using untyped socket handles to identify communication endpoints. Moreover, these APIs are not portable across OS platforms. For example, Win32 lacks Pthreads and has subtly different semantics for sockets and `select`.
- **Increased maintenance effort:** Many ORBs handle portability variations via explicit conditional compilation directives in ORB source code. Using conditional compilation to address platform-specific variations *at all points of use* increases the complexity of the source code, as shown in Section 4. For instance, it is hard to maintain and extend SunSoft IIOp since platform-specific details are scattered throughout the ORB implementation 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 used to implement TAO.

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

Solution → the Wrapper Facade pattern: An effective way to avoid accessing system mechanisms directly is to use the *Wrapper Facade pattern*. This pattern is a variant of the Facade pattern [8]. 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 type-safe, modular, and portable class interfaces that encapsulate lower-level, stand-alone system mechanisms such as sockets, `select`, and Pthreads. In general, the Wrapper Facade pattern should be applied when existing system-level APIs are non-portable and non-type-safe.

Using the Wrapper Facade pattern in TAO: TAO accesses all system mechanisms via the wrapper facades provided by ACE [27]. ACE is an OO framework that implements core concurrency and distribution patterns for communication software. It provides reusable C++ wrapper facades and framework components that are targeted to developers of high-performance, real-time applications and services across a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems (like VxWorks, Chorus, and LynxOS).

Figure 5 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 type-safe OO interfaces. The OO encapsulation provided by ACE alleviates the need for TAO to access the weakly-typed system mechanisms directly. Therefore, C++ compilers can detect type system violations at compile-time rather than at run-time. 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 method calls. Likewise, static methods are used to avoid the overhead of passing a `this` pointer to each invocation.

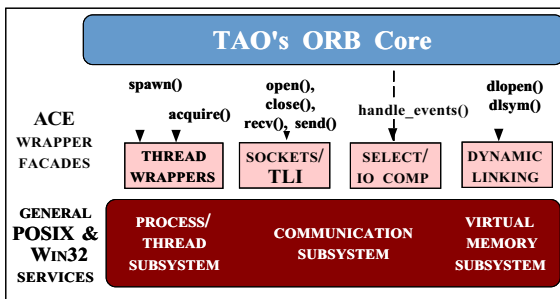


Figure 5: TAO’s Wrapper Facade Encapsulation

Figure 5 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 type-safe OO interfaces. The OO encapsulation provided by ACE alleviates the need for TAO to access the weakly-typed system mechanisms directly. Therefore, C++ compilers can detect type system violations at compile-time rather than at run-time. 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 method calls. Likewise, static methods are used to avoid the overhead of passing a `this` pointer to each invocation.

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

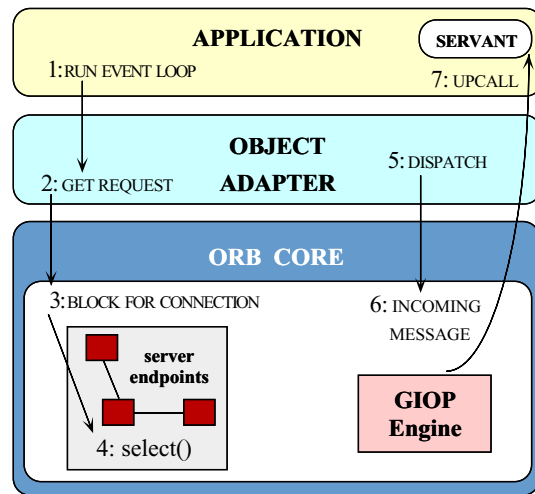


Figure 6: SunSoft IIOP Event Loop

loop by (2) calling `get_request` on the Object Adapter. The `get_request` method then (3) calls class method `block_for_connection` on the `server_endpoint`, which manages all aspects of server-side connection management, ranging from connection establishment to GIOP protocol handling. The ORB remains blocked (4) on `select` until an I/O event such as a connection or a client request occurs. 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: Many ORB Core implementations tightly bind themselves to a single mechanism for demultiplexing events. Relying on one event demultiplexing mechanism is undesirable, however, since no single mechanism is the most efficient

on all platforms. For instance, asynchronous I/O completion ports are very efficient on Windows NT, whereas synchronous threads are the most efficient demultiplexing mechanism on VxWorks.

In addition, many ORB Core implementations tightly couple their event demultiplexing code with the code that performs GIOP protocol processing. For instance, the event demultiplexing logic of SunSoft IOP 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. Therefore, SunSoft IOP's demultiplexing code cannot be reused as a blackbox component by similar communication middleware applications such as HTTP servers [28] or video-on-demand applications. Moreover, if new ORB strategies for threading or Object Adapter request scheduling algorithms are introduced, substantial portions of the the SunSoft IOP ORB Core must be re-written.

How then can an ORB implementation render itself independent of 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* [10]. 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 an application like an ORB Core must handle events from multiple clients concurrently, without committing itself to a single low-level mechanism like `select`.

It is important to note that applying the Wrapper Facade pattern is not sufficient to resolve the problems outlined above. For instance, while a wrapper facade for `select` may improve ORB Core portability somewhat, this pattern does not resolve the need to completely decouple the low-level event demultiplexing logic from the higher-level client request processing logic in the ORB Core.

Using the Reactor pattern in TAO: TAO uses the Reactor pattern to drive the main event loop within its ORB Core, as shown in Figure 7. 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 the Object Adapter. Finally, the Object

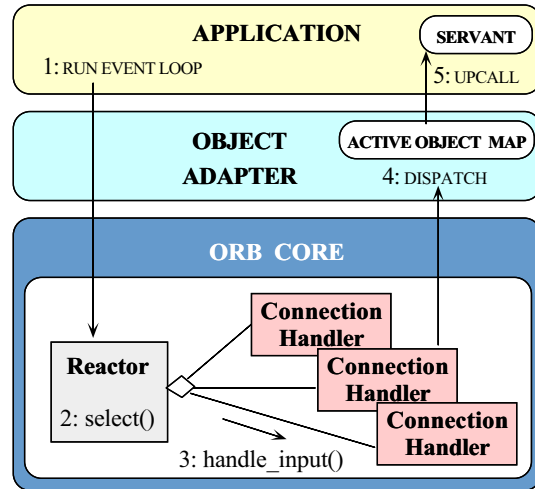


Figure 7: Using the Reactor Pattern in TAO's Event Loop

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` event demultiplexing system call is used on Windows NT, whereas `select` is 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 traffic does not affect the implementation of the Reactor. Finally, unlike the event demultiplexing code in SunSoft IOP, which is tightly coupled to one use-case, the ACE implementation of the Reactor pattern [9] used by TAO has been applied in many other OO event-driven applications ranging from HTTP servers [28] to real-time avionics infrastructure [14].

3.3.3 Managing Connections in an ORB Using Acceptor-Connector Pattern

Context: Connection management is another key responsibility of an ORB Core. For instance, an ORB Core that implements the IOP protocol must establish TCP connections and initialize the protocol handlers for each IOP `server_endpoint`. The reason the CORBA architecture localizes the connection management logic in the ORB Core is to allow application-specific servants to focus solely on processing client requests.

An ORB Core is not *limited* to running over IOP and TCP transports, however. For instance, while TCP can transfer GIOP requests reliably, its flow control and congestion control algorithms may preclude its use as a real-time protocol. Likewise, it may be more efficient to use shared memory as the

transport mechanism when clients and servants are co-located on the same endsystem. Thus, an ideal ORB Core must be flexible in its support of multiple transport mechanisms.

Problem: Well-designed ORBs explicitly decouple the connection management tasks performed by an ORB Core from the request processing performed by an application servant. However, many ORBs still implement their *internal* connection management activities in an inflexible and non-extensible manner. For instance, it is common practice to implement ORB connection management using low-level network APIs like sockets. Likewise, it is also common practice to tightly couple the connection establishment protocol with the communication protocol.

Figure 8 illustrates the connection management structure of SunSoft IIOp. This design hard-codes connection management

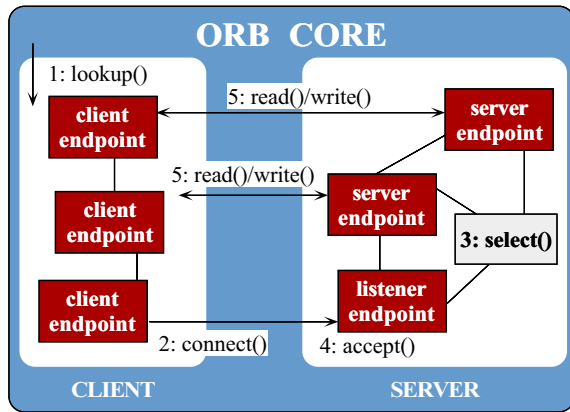


Figure 8: Connection Management in SunSoft IIOp

to use the socket network programming API and also tightly couples the TCP/IP connection establishment protocol with the GIOP communication protocol.

There are two drawbacks to this commonly used design:

1. Too tightly coupled: The client-side of SunSoft IIOp implements a hard-coded connection caching strategy that uses a linked-list of `client_endpoint` objects. As shown in Figure 8, 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 connection endpoints that (4) accept connections and (5) receive requests from clients connected to the server.

Because its data structures and algorithms are closely intertwined, changing any of the steps described above requires substantial alteration of SunSoft IIOp's ORB Core. In

addition, the tight coupling of SunSoft IIOp to the socket API makes it hard to change the underlying transport mechanism. For instance, changing the transport to use shared memory requires re-working both `client_endpoint` and `server_endpoint` components, as well as the low-level GIOP-related routines, because all of these focus on socket descriptors. Thus, porting its ORB Core to new networks such as ATM or different network programming APIs like TLI is intrusive and time consuming.

2. Too inefficient: Many internal ORB strategies can be optimized by allowing both ORB developers and application developers to select appropriate implementations late in the design cycle, *e.g.*, after extensive run-time performance profiling. For example, a multi-threaded real-time client may need to store connection endpoints in thread-specific storage to reduce lock contention and overhead. Similarly, the concurrency strategy for a server might specify that each connection should run in its own thread to eliminate per-request locking overhead. However, SunSoft IIOp makes it difficult to accommodate efficient new strategies since its connection management strategies are hard-coded and tightly bound with its other internal ORB 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* [25]. This pattern decouples connection initialization from the tasks performed once a connection endpoint is initialized. The *Acceptor* component in the 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 setup of connections for GIOP/IIOp communication. Within the client ORB Core, a *Connector* initiates connections to servers in response to a method invocation or explicit binding to an remote object. Within the server ORB Core, one or more *Acceptors* creates a *GIOP Connection Handler* to service each new client connection. *Acceptors* and *ConnectionHandlers* both derive from *EventHandler*, 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 systematically select an appropriate concurrency mechanism, as described in Section 3.3.4.

Figure 9 illustrates the use of Acceptor-Connector strategies in TAO's ORB Core. When a client (1) invokes a

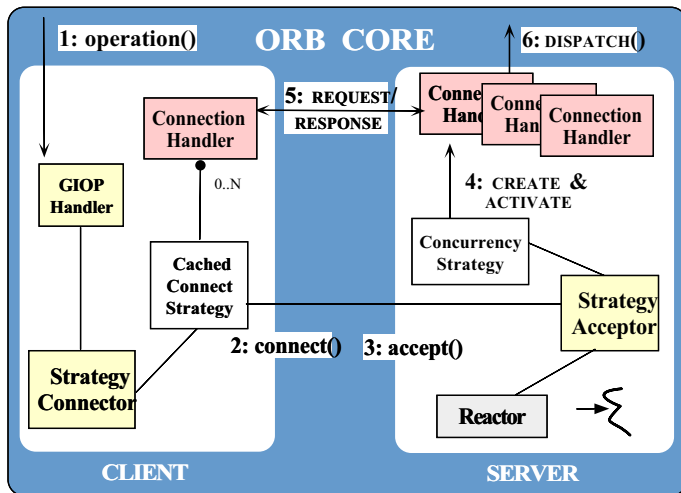


Figure 9: Using the Acceptor-Connector Pattern in TAO's Connection Management

remote operation, it makes a `connect` call through the `Strategy_Connector`. The `Strategy_Connector` (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. Thus, it only creates new connections when all existing connections are already in use. This 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-request, thread-per-connection, thread-per-priority, etc.) described in Section 3.3.4. Once 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.

3.3.4 Simplifying ORB Concurrency using the Active Object Pattern

Context: Once 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 CORBA specification does not address the issue of concurrency within an ORB or a servant, leaving the decision to ORB developers.

It is important to develop efficient concurrent ORBs. For instance, concurrency allows long-running tasks to execute simultaneously without impeding the progress of other tasks. Likewise, preemptive multi-threading is crucial to minimize the dispatch latency of real-time systems [14]. 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 10: a single-threaded Reactive architecture and a thread-per-connection architecture.

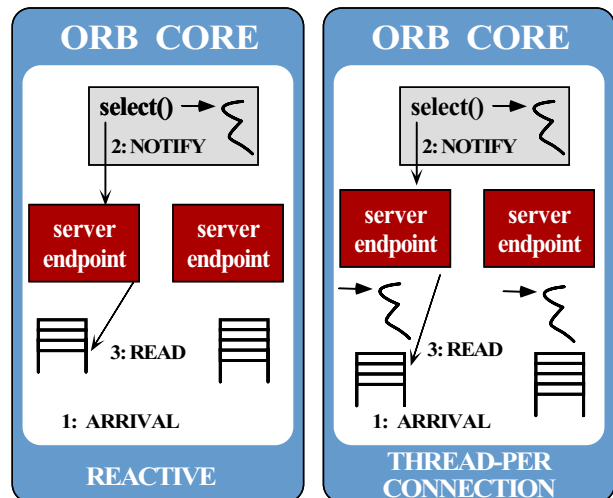


Figure 10: SunSoft IIOp Concurrency Architectures

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.

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. For instance, SunSoft IIOP uses the POSIX Pthreads API. However, there are several drawbacks to this approach:

- **Non-portable:** Threading APIs tend to be very platform-specific. Even industry standards such as POSIX Pthreads are not available on many widely-used OS platforms, including Win32 and VxWorks. Not only is there no direct mapping between APIs, but there is no clear mapping of functionality. For instance, POSIX Pthreads supports thread cancellation, whereas Solaris threads do not. Moreover, Win32 has a thread termination API, but the documentation strongly recommends *not* using it since it does not release thread resources on exit. Moreover, Pthreads itself is highly non-portable since many UNIX vendors implement different versions of the standard.

- **Hard to program correctly:** Programming a multi-threaded ORB is hard since application and ORB developers must ensure that access to shared data is properly serialized in the ORB and in the servants. In addition, the techniques required to robustly terminate servants that execute 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 is an appropriate strategy for short duration, compute-bound requests on a uni-processor. If these external factors change, however, it is important that an ORB can be extended to handle alternative concurrency strategies such as thread-per-request, thread pool, or thread-per-priority.

When an ORB is developed using low-level threading APIs, however, it is hard to extend it 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. Therefore, 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 → the Active Object pattern: An effective way to increase the portability, correctness, and extensibility of ORB concurrency strategies is to apply the *Active Object pattern* [11]. This pattern provides a higher-level concurrency architecture that decouples the thread that initially receives and processes a client request from the thread that ultimately executes the request.

While *Wrapper Facades* provide the genesis for portability, they are simply thin veneers over the low-level system mechanisms. Moreover, a facade's behavior may still vary from platform to platform. Therefore, the Active Object 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 programmers, the Active Object pattern makes it easier to define more portable, flexible, and easy to program ORB concurrency strategies.

In general, the Active Object pattern should be used when an application can be simplified by centralizing the point where concurrency decisions are made. This pattern gives developers the ability to insert decision points – such as whether or not to spawn a thread-per-connection or a thread-per-request – between each request's initial reception and its ultimate execution.

Using the Active Object pattern in TAO: TAO uses the Active Object pattern to transparently allow a GIOP Connection Handler to execute requests either *reactively* by borrowing the Reactor's thread of control or *actively* by running in its own thread of control. The sequence of steps is shown in Figure 11.

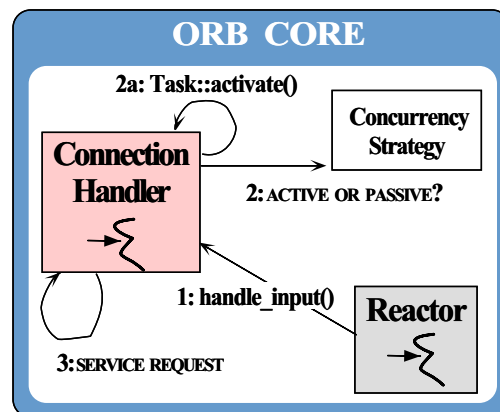


Figure 11: Using the Active Object Pattern to Structure TAO's Concurrency Strategies

The processing shown in Figure 11 is triggered when (1) a Reactor notifies the Connection Handler that an I/O event is pending. Based on the currently configured strategy (e.g., reactive, thread-per-connection, thread-per-request, etc.), the handler (2) determines if it should be active or

passive and acts accordingly. This flexibility is achieved by inheriting TAO's ORB Core connection handling classes from an ACE base class called `Task`. To process a request concurrently, therefore, the handler simply (2a) invokes the `Task::activate` method. This method spawns a new thread and invokes a standard hook method. Whether active or passive, the handler will ultimately (3) process the request.

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

Context: The Active Object pattern allows applications and components in the ORB to operate using a variety of concurrency policies, rather than one enforced by the ORB itself. The primary drawback to concurrency, however, is the need to synchronize access to shared resources, such as operators `new` and `delete`, pointers created by the `CORBA::ORB_init` ORB initialization factory, the `Acceptor` and `Connector` described in Section 3.3.3. A common way to achieve this synchronicity is through the use of mutually-exclusive locks on each resource shared by multiple threads. However, acquiring and releasing these locks can be expensive, often negating any potential performance benefits of concurrency.

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, multi-threaded ORBs often perform no better, or even worse, than single-threaded ORBs due to (1) acquiring/releasing locks and (2) priority inversions that arise when high- and low-priority threads contend for the same locks [29]. In addition, multi-threaded ORBs can be hard to program due to the complex concurrency control protocols required 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. 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 [26].

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 the TAO stores its ORB Core and Object Adapter components, e.g., `Reactor`, `Acceptor`, `Connector`, `POA`, in thread-specific storage. 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 12. Therefore, no additional locking is required to access ORB state.

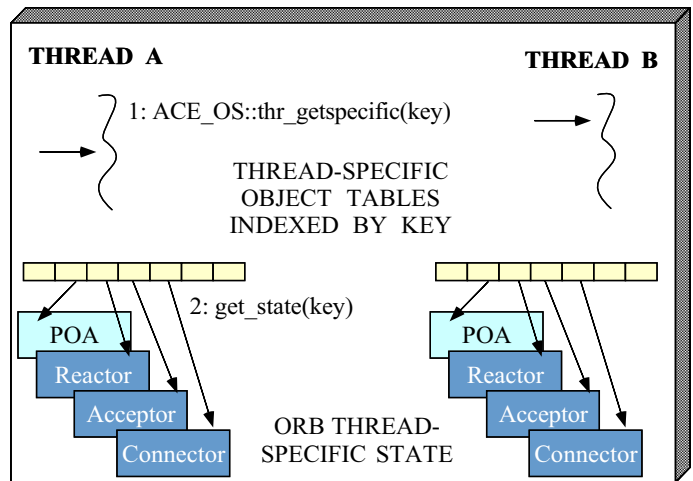


Figure 12: Using the Thread-Specific Storage Pattern TAO

3.3.6 Support Interchangeable ORB Behaviors with the Strategy Pattern

Context: The alternative concurrency architectures described in 3.3.4 are just one of the many strategies that an extensible ORB like TAO must support. In general, extensible ORBs must support multiple request demultiplexing and scheduling strategies in their Object Adapters, as well as multiple connection establishment, request transfer, and concurrent request processing strategies in their ORB Cores.

Problem: Conventional ORBs only provide static, non-extensible strategies. For instance, the strategies in SunSoft IOP are configured in the following ways:

- **Preprocessor macros:** Some strategies are determined by the value of preprocessor macros. For example, since threading is only available on selected platforms, SunSoft IOP uses conditional compilation to select the appropriate concurrency architecture.
- **Command-line options:** Other strategies are controlled by the presence or absence of flags on the command-line. For instance, the `-t` command-line option enables the thread-per-connection concurrency strategy in the SunSoft IOP ORB Core.

While these two configuration approaches are widely used in conventional ORBs, they are very 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 completely divorced 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. These restrictions make it hard to develop highly extensible ORBs built 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* [8]. This pattern factors out similarity 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 only access specialized behaviors through common base class interfaces. In general, the Strategy pattern should be used when an application’s behavior can be configured using multiple strategies that must be interchanged seamlessly.

Using the Strategy Pattern in TAO: TAO uses a variety of communication, concurrency, demultiplexing, real-time scheduling and dispatching strategies to factor out behaviors that are typically hard-coded in conventional ORBs. Several of these strategies are illustrated in Figure 13. For instance, TAO supports multiple request demultiplexing strategies (*e.g.*, perfect hashing vs. active demultiplexing [24]) and scheduling strategies (*i.e.*, FIFO vs. rate monotonic vs. earliest deadline first [14]) 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 Active Objects) in its ORB Core.

3.3.7 Consolidate ORB Strategies Using the Abstract Factory Pattern

Context: There are dozens of potential strategy variants in TAO. Table 1 shows a simple example of the strategies used to create two configurations of TAO. One is an avionics application with hard real-time requirements [14] and the other is an electronic medical imaging application [15] 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.

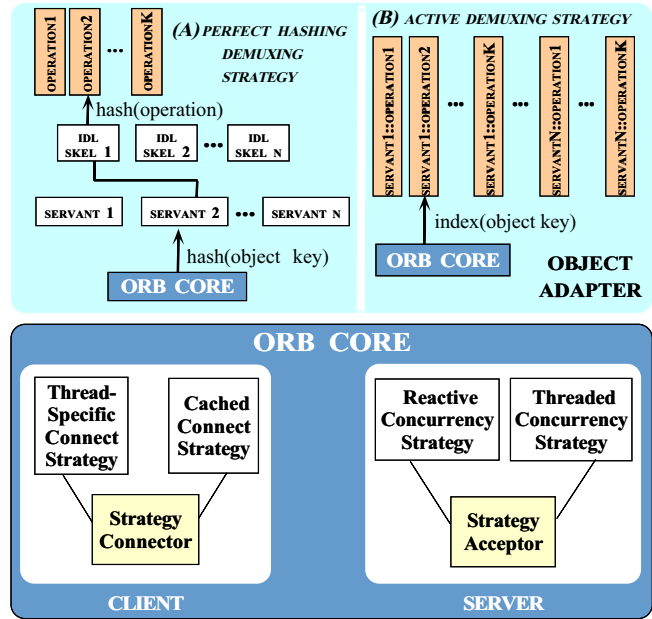


Figure 13: Strategies in TAO

Problem: An undesirable side-effect of using the Strategy pattern extensively in a complex application like an ORB is that maintenance and extensibility become hard to manage for the following reasons:

- **Software complexity:** ORB source code can become littered with direct references to strategy types. Many independent strategies must act in harmony to provide a comprehensive solution to particular application domains, such as real-time avionics. However, identifying these strategies individually by name 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 strategies to interact compatibly. For instance, the FIFO strategy for scheduling requests shown in Table 1 might 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 only feasible for systems that statically configure all servants off-line.

How can a highly-configurable ORB reduce the complexities required in managing its myriad of strategies, as well as

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

Table 1: Example Applications and Their ORB Strategy Configurations

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* [8]. This pattern provides a single access point that integrates all strategies used to configure an ORB. Concrete subclasses then aggregate semantically compatible application-specific or domain-specific strategies, which can be replaced wholesale in semantically meaningful ways. In general, the Abstract Factory pattern should be used when an application needs to consolidate the configuration of many strategies, each having multiple variations.

Using the Abstract Factory pattern in TAO: All of TAO’s ORB strategies are consolidated into two abstract factories implemented as Singletons [8]. One factory encapsulates client-specific strategies, while the factory shown in Figure 14 encapsulates server-specific strategies. These abstract factories

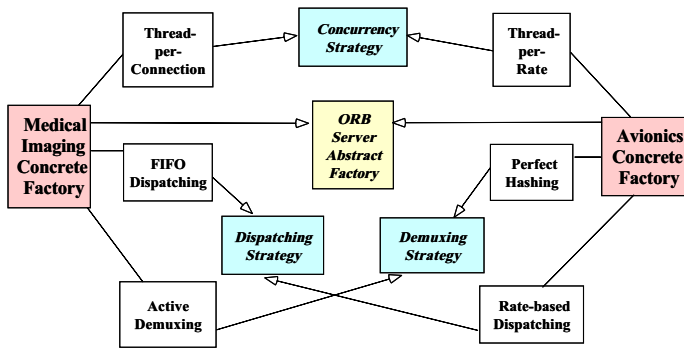


Figure 14: Factories used in TAO

encapsulate concurrency strategies in both the client and the server, and request demultiplexing, scheduling, and dispatch strategies in the server. By using the Abstract Factory pattern, TAO can configure different ORB personalities conveniently and consistently.

3.3.8 Dynamically Configure ORBs with the Service Configurator Pattern

Context: Although the cost of many resources such as memory continues to drop, ORBs must still avoid excessive con-

sumption of finite system resources. This is particularly important for embedded real-time systems that require small memory footprints. Likewise, many applications can benefit from an ability to extend ORBs *dynamically* by allowing their strategies to be configured at run-time.

Problem: Although the Strategy and Abstract Factory patterns make it easier to customize ORBs for specific application requirements and system characteristics, these patterns can cause the following problems for extensible ORBs:

- **High resource utilization:** Widespread use of the Strategy pattern can substantially increase 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 IOP does permit applications to vary certain ORB strategies at run-time. However, the different strategies must be configured statically into SunSoft IOP at compile-time. Moreover, as the number of alternatives increases, so does the amount of code required to implement them. For instance, Figure 15 illustrates SunSoft IOP’s approach to varying the concurrency strategy.

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. Using this technique to configure complex extensible ORBs complicates maintenance, increases

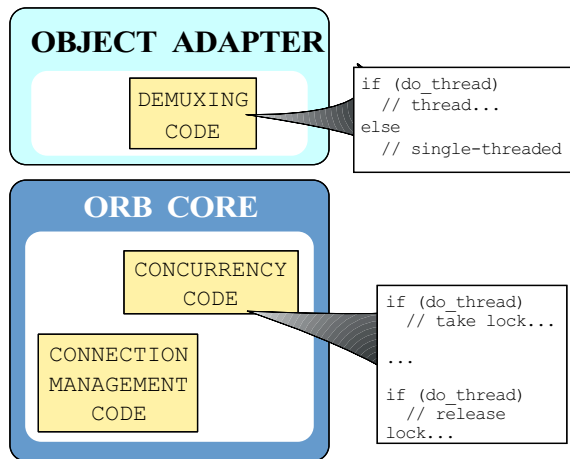


Figure 15: SunSoft IOP Hard-coded Strategy Usage

system resource utilization, and leads to unavoidable system downtime to add or change existing components.

How then does an ORB implementation reduce the “overly-large, overly-static” side-effect of the widespread use of the Strategy and Abstract Factory patterns?

Solution → the Service Configurator pattern: An effective way to enhance the dynamism of an ORB is to apply the *Service Configurator pattern* [22]. This pattern uses explicit dynamic linking [23] 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 or run-time. Widely available explicit dynamic linking mechanisms include the `dlopen/dlsym/dlclose` functions in SVR4 UNIX [30] and the `LoadLibrary/GetProcAddress` functions in the WIN32 subsystem of Windows NT [31]. The ACE wrapper facades provide a portable encapsulation of these functions.

By using the Service Configurator pattern, the *behavior* of ORB strategies are decoupled from *when* implementations of these strategies are configured into an ORB. For instance, ORB strategies can be linked into an ORB from DLLs at compile-time, installation-time, or even during run-time. Moreover, this pattern can reduce the memory footprint of an ORB by allowing application developers to dynamically link only those strategies that are necessary for a specific ORB personality.

In general, the Service 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 Service Configurator pattern in TAO: TAO uses the Service Configurator pattern to configure abstract fac-

ories at run-time that contain the desired strategies. TAO’s initialization code uses the dynamic linking mechanisms provided by the OS and encapsulated by the ACE wrapper facades to link in the appropriate factory for a particular use-case. This design allows applications to select the personality of a particular ORB at run-time. It also allows TAO’s behavior to be tailored to specific environments and application requirements without requiring access to the ORB source code.

Figure 16 shows two factories tuned for different application domains – Avionics and Medical Imaging. In this partic-

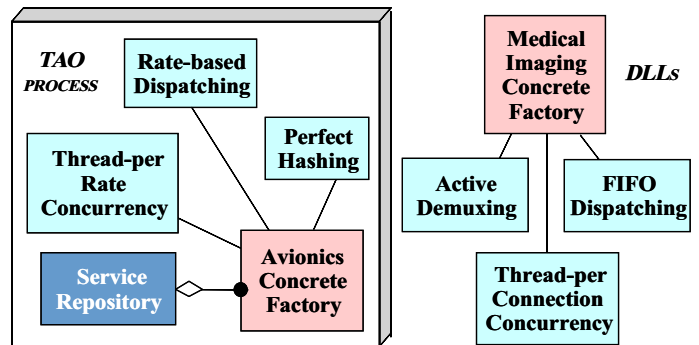


Figure 16: Using the Service Configurator Pattern in TAO

ular configuration, the Avionics factory is currently installed in the process. Applications using this ORB configuration will be processed with a particular set of ORB concurrency, demultiplexing, and dispatching strategies. In contrast, the Medical Imaging factory resides in a DLL outside of the current ORB process. To support a different configuration of the ORB this factory could be dynamically installed when the ORB process is first initialized.

4 Evaluating the Contribution of Patterns to ORB Middleware

Table 2 summarizes the mapping between ORB design challenges and the patterns we used to resolve these challenges. This table focuses on the forces resolved by individual patterns. However, TAO also benefits from the collaborations among *multiple* patterns (shown in Figure 3). For example, the Acceptor and Connector patterns utilize the Reactor pattern to notify them when connection events occur in the OS.

Often, patterns must collaborate to alleviate drawbacks that arise from applying them in isolation. For instance, the primary motivation for using the Abstract Factory pattern in TAO is to avoid the potential complexity created by using the Strategy pattern extensively in an ORB. Although the Strategy pattern simplifies the effort required to tailor an ORB for specific

Forces	Resolving Pattern(s)
Abstracting low-level system calls	Wrapper Facade
ORB event demultiplexing	Reactor
ORB connection management	Acceptor, Connector
Efficient concurrency models	Active Object
Pluggable strategies	Strategy
Group similar initializations	Abstract Factory
Dynamic run-time configuration	Service Configurator

Table 2: Summary of Forces and their Resolving Patterns

application requirements and network/endsystem characteristics, it is tedious and error-prone to manage large numbers of strategy interactions manually. Therefore, the Service Configurator pattern can be used in conjunction with the Strategy and Abstract Factory patterns to update the strategies used by TAO without modifying existing code, recompiling or statically re-linking existing code, or terminating and restarting an existing ORB and its application servants.

4.1 Where’s the Proof?

Implementing TAO using patterns yielded some expected and some unexpected improvements in software reusability and maintainability. The results are summarized in Table 3. This table compares the following metrics for TAO and SunSoft IOP: (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, and (3) the average McCabe Cyclometric Complexity metric $v(G)$ [32] 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.

As shown in the results, the use of patterns in TAO significantly reduced the amount of *ad hoc* code and the complexity of certain operations. In particular, 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 IOP 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 17. This figure illus-

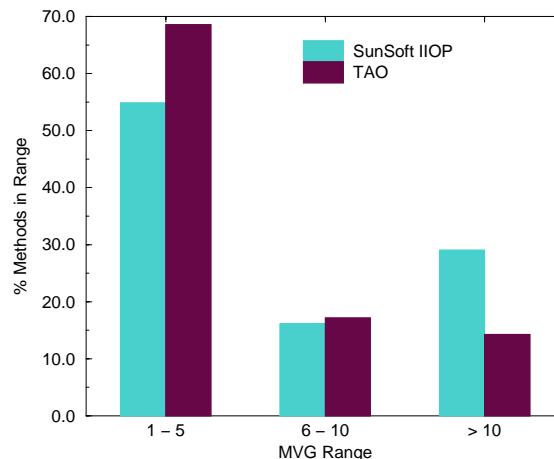


Figure 17: Distribution of $v(G)$ Over ORB Methods

trates the distribution of $v(G)$ over the percentage of affected methods in TAO. As shown by 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. In contrast, while SunSoft IOP has a substantial percentage (55%) of its methods in that range, the bulk of the remaining methods (29%) have $v(G)$ greater than 10. The reason for the difference is that SunSoft IOP 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 IOP methods were decomposed into smaller methods when integrating the patterns. The bulk of TAO’s methods—86% to be exact—have $v(G)$ under 10, and of that nearly 70% lie between 1 and 5. Of the relatively few (14%) methods in TAO with $v(G)$ greater

ORB Task	TAO			SunSoft IIOP		
	# 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

than 10, most are largely unchanged from the original SunSoft IIOP TypeCode interpreter. Not only does the use of monolithic methods increase the maintenance effort, it also degrades the TypeCode interpreter’s performance due to reduced processor cache hits [33]. Therefore, we plan to experiment with the application of other patterns, such as *Command* and *Template Method* [8], to simplify and optimize these monolithic methods into smaller, more cohesive methods. There are a few methods with $v(G)$ greater than 10 which are not part of the TypeCode interpreter, and they will likely remain that way. Sometimes solving complex problems involves writing complex code; at such times, localizing complexity is a reasonable recourse.

4.2 What are the Benefits?

In general, the use of patterns in TAO provided the following benefits:

- **Increased extensibility:** Patterns like Abstract Factory, Strategy, and Service Configurator make it much easier to re-configure TAO for a particular application domain by allowing extensibility to be “designed into” an ORB. In contrast, middleware that lacks these patterns is significantly harder to develop and extend. This article illustrated how design patterns were applied to an existing ORB (SunSoft IIOP) to make it more extensible.

- **Enhanced maintenance:** Design patterns are essential to capture and articulate the design rationale for complex structures in an ORB. Patterns help to demystify and motivate the structure of an ORB by describing its architecture in terms of design forces that recur in many software systems. The expressive power of patterns enabled us to convey the design of complex software systems like TAO. Thus, the patterns presented in this article help to improve the maintainability of ORB middleware by reducing software complexity, as shown in Figure 17.

- **Increased portability and reuse:** Constructing our ORB atop an OO communication framework like ACE simplified the effort required to port TAO to various real-time platforms. Most of the porting effort is 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.

4.3 What are the Liabilities?

The use of patterns can 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 [33] and demultiplexing overhead [24]) to minimize performance overhead. As a result, TAO is substantially faster than the original hard-coded SunSoft IIOP [33].

- **Additional external dependencies:** Whereas SunSoft IIOP only depends on system-level interfaces and libraries, TAO now depends on 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.

5 Concluding Remarks

This article presents a case study that shows how we applied patterns to enhance the extensibility and maintainability of TAO, a dynamically configurable, real-time ORB. We found qualitative and quantitative evidence that the use of patterns 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 using patterns in TAO is that the systematic application of the patterns presented in this paper significantly improved the decoupling and object-oriented structure of the TAO ORB. All the patterns utilized were applied in roughly the same order that they appear in Section 3.3. Each stage built upon prior stages. This process revealed new insights on which patterns could be applied and how they might be applied in subsequent stages.

Another—perhaps even more important—result is that through the careful application of these patterns, we not only sculpted a better ORB, but also defined a better vocabulary to discuss ORB middleware designs. This vocabulary is the first “enabling” step to demystify the internals of an ORB. As we continue to learn about ORBs and the patterns of which they are composed, we expect that this vocabulary will grow and evolve.

The source code for ACE and TAO is freely available at www.cs.wustl.edu/~schmidt/TAO.html.

Acknowledgements

We would like to thank Frank Buschmann, Hans Rohnert, and Michael Stal for their extensive comments on this paper.

References

- [1] R. Johnson, “Frameworks = Patterns + Components,” *Communications of the ACM*, vol. 40, Oct. 1997.
- [2] S. Landis and S. Maffeis, “Building Reliable Distributed Systems with CORBA,” *Theory and Practice of Object Systems*, Apr. 1997.
- [3] S. Vinoski, “CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments,” *IEEE Communications Magazine*, vol. 14, February 1997.
- [4] 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.
- [5] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.
- [6] D. Box, *Understanding COM*. Addison-Wesley, Reading, MA, 1997.
- [7] A. Wollrath, R. Riggs, and J. Waldo, “A Distributed Object Model for the Java System,” *USENIX Computing Systems*, vol. 9, November/December 1996.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- [9] 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.
- [10] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.
- [11] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.
- [12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.
- [13] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers,” *Computer Communications*, to appear, 1998.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] A. Gokhale and D. C. Schmidt, “Design Principles and Optimizations for High-performance ORBs,” in *12th OOPSLA Conference*, poster session, (Atlanta, Georgia), ACM, October 1997.
- [18] D. C. Schmidt, R. Bector, D. Levine, S. Mungee, and G. Parulkar, “An ORB Endsystem Architecture for Statically Scheduled Real-time Applications,” in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.
- [19] D. C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible and Maintainable ORB Middleware,” *Communications of the ACM*, to appear, 1998.
- [20] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zyk, and R. Johnston, “Real-Time CORBA,” in *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, (Montréal, Canada), June 1997.

- [21] Z. D. Dittia, G. M. Parulkar, and J. Jerome R. Cox, "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), IEEE, April 1997.
- [22] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.
- [23] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.
- [24] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.
- [25] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.
- [26] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently," in *The 4th Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.
- [27] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.
- [28] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the 2nd Global Internet Conference*, IEEE, November 1997.
- [29] D. C. Schmidt, S. Mungee, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time ORB Core Architectures," in *Submitted to the Fourth IEEE Real-Time Technology and Applications Symposium*, (San Francisco, CA), IEEE, December 1997.
- [30] 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.
- [31] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.
- [32] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, Dec. 1976.
- [33] A. Gokhale and D. C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance," in *Hawaiian International Conference on System Sciences*, January 1998.