Patterns, Frameworks, and Middleware: Their Synergistic Relationships

Douglas C. Schmidt
douglas.c.schmidt@vanderbilt.edu
Electrical Engineering & Computer Science
Vanderbilt University
Nashville Tennessee, USA*

Frank Buschmann
Frank.Buschmann@siemens.de
Corporate Research
Siemens, AG
Munich, Germany

Abstract

The knowledge required to develop complex software has historically existed in programming folklore, the heads of experienced developers, or buried deep in the code. These locations are not ideal since the effort required to capture and evolve this knowledge is expensive, time-consuming, and error-prone. Many popular software modeling methods and tools address certain aspects of these problems by documenting how *a system is designed. However, they only support limited portions of software development and do not articulate* why *a system is designed in a particular way, which complicates subsequent software reuse and evolution.*

Patterns, frameworks, and middleware are increasingly popular techniques for addressing key aspects of the challenges outlined above. Patterns codify reusable design expertise that provides time-proven solutions to commonly occurring software problems that arise in particular contexts and domains. Frameworks provide both a reusable product-line architecture [1] – guided by patterns – for a family of related applications and an integrated set of collaborating components that implement concrete realizations of the architecture. Middleware is reusable software that leverages patterns and frameworks to bridge the gap between the functional requirements of applications and the underlying operating systems, network protocol stacks, and databases. This paper presents an overview of patterns, frameworks, and middleware, describes how these technologies complement each other to enhance reuse and productivity, and then illustrates how they have been applied successfully in practice to improve the reusability and quality of complex software systems.

1 Introduction

Emerging trends and challenges. Despite significant advances in computers, networks, programming languages, and software methodologies, developing quality applications on time and within budget remains hard. Some problems stem from the growing demands placed on software, such as requirements to provide predictable, reliable, scalable, and secure quality of service (QoS) simultaneously. Other problems stem from the propensity to rediscover and reinvent core software artifacts, such as programming languages, operating systems, networking protocols, and software component libraries. Moreover, the heterogeneity of hardware architectures, diversity of operating system (OS) and network platforms, and stiff global competition makes it increasingly infeasible to build high-quality software from scratch.

In today's time-to-market-driven environments, building quality software in cost-effective manner requires the *systematic reuse* of successful software models, designs, and implementations that have already been developed and tested. Unlike opportunistic reuse (in which developers simply cut and paste code from existing programs to create new ones), systematic reuse is an intentional and concerted effort to create and apply multiuse software artifacts throughout an organization. In a well-honed systematic reuse process, each new project leverages time-proven designs and implementations, mostly just adding new code that is specific to a particular application, and only refactoring existing software architectures and designs when they become inadequate to cover the evolving business cases and variability in the supported domains. Systematic reuse is essential to increase software productivity and quality by breaking the costly cycle of rediscovering, reinventing, and revalidating common software artifacts. Throughout the rest of the paper when we use the term "reuse" we therefore always mean "systematic reuse."

The skills required to develop, deploy, and support reusable software have traditionally been a "black art" practiced only by expert developers and architects. Moreover, these technical impediments to reuse are often exacerbated by a myriad of nontechnical impediments, such as organizational, economic, administrative, political, sociological, and psychological factors. It's therefore not surprising that significant levels of software reuse have been slow to materialize in many projects and

*This work was supported in part by DARPA, NSF, and Siemens.

organizations.

Solution approach → Middleware, framework, and patterns. *Middleware* [2] is software that can significantly increase reuse by providing readily usable, standard solutions to common programming tasks, such as persistent storage, (de)marshaling, message buffering and queueing, request demultiplexing, and concurrency control. Developers who use middleware can therefore focus primarily on application-oriented topics, such as business logic, rather than wrestling with tedious and error-prone details associated with programming infrastructure software using lower-level OS APIs and mechanisms.

Over the past decade a number of middleware standards have emerged and matured. Some of these standards, such as the Common Object Request Broker Architecture (CORBA), are open systems sanctioned by international organizations, such as the Object Management Group (OMG). Other middleware standards, such as Java virtual machines, Jave 2 Enterprise Edition (J2EE), and .NET, have emerged from industry consortia and market leaders.

Crucial to the success of standard middleware are the *patterns* [3, 4, 5] and *frameworks* [6, 7] that reify the knowledge of how to develop and apply the middleware and applications that run atop it. Patterns support the reuse of design expertise by articulating the static and dynamic aspects of successful solutions to problems that arise when building software in a particular context. Frameworks are concrete realizations of groups of patterns that enable reuse of code by (1) capturing the common abstractions of an application domain – both their structure and mechanisms – while (2) yielding control of application-specific structure and behavior to application developers.

During the past decade a number of influential R&D efforts [4, 8, 5, 9, 10, 11, 12, 13] have focused on documenting patterns and developing frameworks that enable the effective development and reuse of middleware. As a result, middleware is now commonly developed using frameworks based on strategized composition and optimization patterns. These patterns and frameworks guide the integration and configuration of middleware that can meet the functional and QoS requirements of particular application domains more effectively than can realistically be developed manually from scratch in time-to-market driven environments.

The relationship between middleware, frameworks, and patterns is highly synergistic. For example, patterns help guide framework design and use, thereby reducing software development effort and training costs. In turn, frameworks can be used to develop middleware for product-line architectures [1], whose interfaces then provide application software with a simpler facade to access the powerful (and complex) internal component structure of the frameworks. Likewise, frameworks and patterns support the development of product-line architectures for specific application domains, such as warehouse management or hot rolling mill process automation [14], that apply particular types of middleware. A contribution of this paper is to illustrate by examples how these three technologies relate and complement each other.

Paper organization. The remainder of this paper is organized as follows: Section 2 presents an taxonomy of the layers of middleware; Sections 3 and 4 describe the key properties of frameworks and patterns, respectively; Section 5 then illustrates the synergies between all three technologies via a case study of how frameworks and patterns have been applied in practice to develop middleware for distributed real-time and embedded systems; Section 6 summarizes promising areas of future R&D; and Section 7 presents concluding remarks.

2 An Overview of Middleware

Some of the most successful techniques and tools devised to enhance the reuse of software center on middleware that helps manage the complexity and heterogeneity of distributed applications. This *distributed computing middleware* (henceforth referred to simply as *middleware*) provides reusable software that functionally bridges the gap between (1) end-to-end application functional requirements and (2) the lower-level operating systems, networking protocol stacks, databases, and hardware devices. Middleware provides reusable capabilities whose qualities are critical to help simplify and coordinate how application software is connected and how it interoperates.

Just as networking protocol stacks can be decomposed into multiple layers, such as the physical, data-link, network, transport, session, presentation, and application layers, so to can middleware be decomposed into the multiple layers [2] shown in Figure 1 and described in the remainder of this section.

Host infrastructure middleware encapsulates and enhances native OS mechanisms to create reusable event demultiplexing, interprocess communication, concurrency, and synchronization objects, such as reactors; acceptors, connectors, and service handlers; monitor objects; active objects; and service configurators [9, 10]. By encapsulating the peculiarities of particular operating systems, these reusable objects help eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining application software via low-level OS programming APIs, such as Sockets or POSIX pthreads.

Common host infrastructure middleware includes the Sun Java Virtual Machine (JVM) and Microsoft's Common Language Runtime (CLR), which provide platform-independent ways of executing code by abstracting the differences between operating systems and CPU architectures. The ADAPTIVE Communication Environment (ACE) [9, 10] is portable

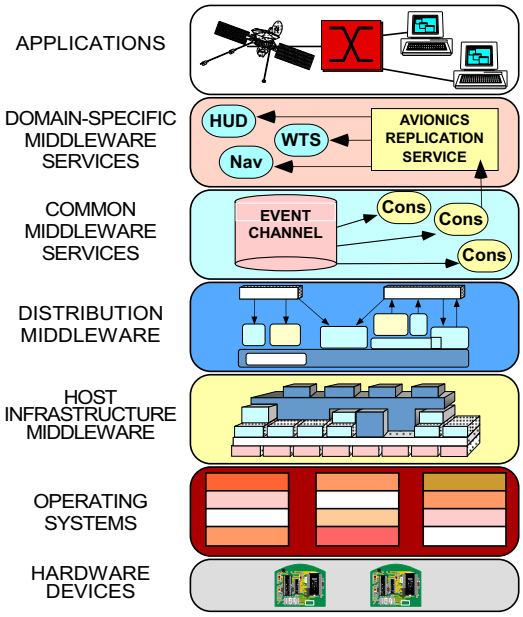

Figure 1: Middleware Layers in Context

C++ host infrastructure middleware that encapsulates native operating system capabilities, such as connection establishment, event demultiplexing, interprocess communication, (de)marshaling, dynamic configuration of application components, concurrency, and synchronization.

Distribution middleware defines higher-level distributed programming models whose reusable APIs and objects automate and extend the native OS mechanisms encapsulated by host infrastructure middleware. Distribution middleware enables clients to program applications by invoking operations on target objects without hard-coding dependencies on their location, programming language, OS platform, communication protocols and interconnects, and hardware.

At the heart of distribution middleware are request brokers, such as The OMG's Common Object Request Broker Architecture (CORBA) and Sun's Java Remote Method Invocation (RMI). The request brokers allow objects to interoperate across networks regardless of the platform on which they are deployed. SOAP is an emerging distribution middleware technology based on an XML-based protocol that allows applications to exchange structured and typed information on the Web using various Internet protocols, such as HTTP, SMTP, and MIME.

Common middleware services augment distribution middleware by defining higher-level domain-independent reusable services that allow application developers to concentrate on programming business logic, without the need to write the "plumbing" code required to develop distributed applications via lower-level middleware directly. For example, common middleware service providers bundle transactional behavior, security, and database connection pooling and threading into reusable components, so that application developers no longer need to write code that handles these tasks.

Whereas distribution middleware focuses largely on managing end-system resources in support of an object-oriented distributed programming model, common middleware services focus on allocating, scheduling, and coordinating various resources throughout a distributed system using a component programming and scripting model. Developers can reuse these component services to manage global resources and perform common distribution tasks that would otherwise be implemented in an *ad hoc* manner within each application. The form and content of these services will continue to evolve as the requirements on the applications being constructed expand.

Examples of common middleware services include the OMG's CORBA Common Object Services, such as event notification, logging, multimedia streaming, persistence, security, global time, real-time scheduling, fault tolerance, concurrency control, and transactions. Likewise, Sun's Enterprise Java Beans (EJB) technology and Microsoft's .NET allows developers to create n-tier distributed systems by linking a number of pre-built software service components without having to write much code manually.

Domain-specific middleware services are tailored to the requirements of particular domains, such as telecom, e-commerce, health care, process automation, or aerospace. Unlike the other three middleware layers discussed above that provide broadly reusable "horizontal" mechanisms and services, domain-specific middleware services are targeted at "vertical" markets and product-line architectures. Since they embody knowledge of a domain, moreover, reusable domain-specific middleware services have the most potential to increase the quality and decrease the cycle-time and effort required to develop particular types of application software.

An example of domain-specific middleware services includes the Siemens Medical Solutions Group has developed `syngo` (`www.syngo.com`), which is a product-line architecture that is both an integrated collection of domain-specific middleware services, as well as an open and dynamically extensible application server platform for medical imaging tasks and applications, including ultrasound, mammography, angiography, computer tomography, magnetic resonance, and nuclear medicine. The Boeing Bold Stroke [15, 16] product-line architecture is another example of domain-specific middleware. Bold Stroke uses COTS hardware and middleware to produce a non-proprietary, standards-based component architecture for military avionics mission computing capabilities, such as navigation, display management, sensor management and situational awareness, data link management, and weapons control.

As the products associated with the reusable middleware

layers summarized above have matured and become pervasive throughout the industry, the total amount of software that application developers must write has shrunk substantially. However, reduction in application developer effort also implies more work for middleware developers since they become responsible for an increasing number of software layers. While some believe that the techniques used to develop reusable middleware are simply implementation details, in practice these choices have a significant impact on other key middleware qualities, such as its affordability, extensibility, flexibility, portability, predictability, efficiency, reliability, scalability, and trustworthiness.

As a result, middleware developers must themselves apply more powerful reuse technologies to simplify the evolution of their software layers. Not surprisingly, frameworks and patterns are increasingly applied to improve the reusability and quality of both middleware and application software. The remainder of this paper presents overviews of frameworks and patterns and then shows how these technologies can be combined with middleware to enhance software reuse. It is important to note, however, that the reuse potential of frameworks and patterns is *not* limited to middleware!

3 Overview of Frameworks

As outlined in the previous section, in today's competitive, fast-paced computing industry, successful middleware and application software must possess (1) **affordability**, to ensure that the total ownership costs of software acquisition and evolution are not prohibitively high, (2) **extensibility**, to support successions of quick updates and additions to address new requirements and take advantage of emerging markets, (3) **flexibility**, to support a growing range of multimedia data types, traffic flows, and end-to-end QoS requirements, (4) **portability**, to reduce the effort required to support applications on heterogeneous OS platforms and compilers, (5) **predictability** and **efficiency**, to provide low latency to delay-sensitive real-time applications, high performance to bandwidth-intensive applications, and usability over low-bandwidth networks, such as wireless links, (6) **reliability**, to ensure that applications are robust and tolerant of faults, (7) **scalability**, to enable applications to handle large numbers of clients simultaneously, and (8) **trustworthiness**, to ensure integrity, confidentiality, and availability in distributed systems.

When software is developed monolithically, *i.e.*, as tightly coupled clumps of functionality that are not organized modularly or hierarchically, it is hard to achieve these qualities. *Frameworks* [6, 7] have emerged as a powerful technology for developing and reusing middleware and application software that possess the qualities listed above. Figure 2 illustrates the three characteristics of frameworks [17] described below that help them to achieve these qualities.[1]

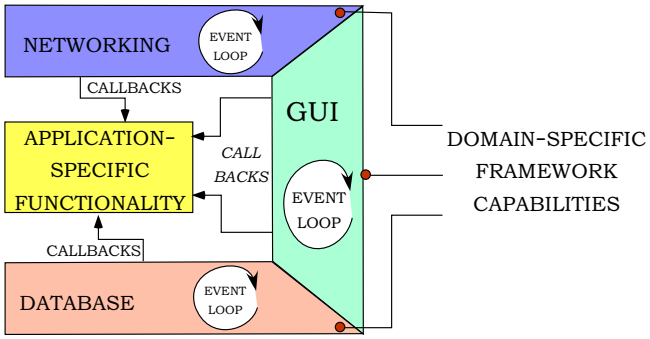

Figure 2: Relationships Between Framework Components

- **A framework exhibits "inversion of control" at run time via callbacks** to component hook methods after the occurrence of an event, such as a mouse click or data arriving on a network connection. When an event occurs, the framework calls back to a virtual hook method in a pre-registered component, which then performs application-defined processing in response to the event. The virtual hook methods in the components decouple the application software from the reusable framework software, which makes it easier to extend and customize the applications as long as the interaction protocols and QoS properties are not violated.
- **A framework provides an integrated set of domain-specific structures and functionality.** Reuse of software depends largely on how well frameworks model the commonalities and variabilities [19] in application domains, such as business data processing, telecom call processing, graphical user interfaces, or real-time middleware. By leveraging the domain knowledge and prior efforts of experienced developers, frameworks embody common solutions to recurring application requirements and software design challenges that need not be recreated and revalidated for each new application.
- **A framework is a "semi-complete" application** that programmers can customize to form complete applications by extending reusable components in the framework. In particular, frameworks help abstract the canonical control flow of applications in a domain into product-line architectures and families of related components. At runtime, these components can collaborate to integrate customizable application-independent reusable code with customized application-defined code.

Since frameworks exhibit inversion of control, they can sim-

[1] While there are subtle technical distinctions between frameworks and components [18, 10], we subsume components into the discussion of frameworks below.

plify application design because the framework—rather than the application—runs the event loop to detect events, demultiplex events to event handlers, and dispatch hook methods on the handlers that process the events. Since frameworks reify the key roles and relationships of components in application domains, the amount of reusable code increases and the amount of code rewritten for each application decreases dramatically. Since a framework is a semi-complete application, it enables larger-scale reuse of software than can be achieved by reusing individual components or stand-alone functions.

Developers in certain domains have applied frameworks successfully for several decades. Early frameworks, such as MacApp, X-windows, and Interviews, originated in the domain of graphical user interfaces (GUIs). Java Swing and Microsoft Foundation Classes (MFC) are contemporary GUI frameworks that are widely used to create graphical applications on PC platforms. The broad adoption of reusable GUI frameworks has yielded many productivity and quality benefits for business and desktop applications.

Building upon their success in the GUI domain, frameworks are now being applied to many new and more complex domains [6, 7, 20]. For example, ACE and TAO described in Section 5.1 are frameworks for host infrastructure and distribution middleware, JBoss and BEA's WebLogic Server are component frameworks for application servers, and many recent R&D efforts, such as the Open Grid Service Infrastructure (OGSI), focus on frameworks for web services. There are also frameworks for specific application domains, such as SAP, and application frameworks for medical imaging systems, such as the Syngo platform. These frameworks are increasingly aligning with the layers of middleware standards and domain-specific product-line architectures discussed in Section 2.

4 Overview of Patterns

Developers of middleware and application software must address many challenges related to complex design and programming topics, such as persistence, data organization, connection management, service initialization, distribution, concurrency control, flow control, error handling, event loop integration, and dependability. Many of these challenges occur repeatedly in many applications and domains. Until the mid-1990's, the knowledge needed to resolve these challenges existed largely in programming folklore, the heads of expert researchers and developers, or buried deep in complex source code. These locations are not ideal because

- Discovering patterns from source code is expensive and time-consuming since it is hard to separate the essence of a design from its implementation details.
- If the insights and rationale of experienced designers are not documented, they will be lost over time and thus cannot be used to guide subsequent software evolution.
- Without guidance from earlier work, software developers must engineering complex systems from the ground up, rather than by reusing proven solutions.

Over the past decade, experienced software developers and architects have helped to address these problems by creating a body of literature that documents the following types of reusable knowledge:

- **Design patterns** [3], which provide a scheme for refining the elements of a software system and the relationships between them, and describe a common structure of communicating elements that solves a general design problem within a particular context.
- **Architectural patterns** [4], which express the fundamental, overall structural organization of software systems and provide a set of predefined subsystems, specify their responsibilities, and include guidelines for organizing the relationships between them.
- **Pattern languages** [21], which weave together a web of related patterns to define a vocabulary for talking about software development problems and provide a process for the orderly resolution of these problems.

Patterns help enhance reuse by capturing and reusing the static and dynamic structure and collaboration of key participants in software designs. They are particularly useful for documenting recurring micro-architectures, which are abstractions of software components that experienced developers apply to resolve common design and implementation problems. By studying and applying patterns and pattern languages, developers can often escape traps and pitfalls that have been avoided traditionally only via long and costly apprenticeships [22]. Patterns also raise the level of discourse in project design and programming activities, which helps improve team productivity and software quality.

Figure 3 illustrates the relationships amongst a pattern language [5] that addresses service access and configuration, event handling, interprocess communication, concurrency, and synchronization dimensions in various networked application domains, such as online financial services, remote process control, avionics mission computing, and telecommunications.[2] The relationships in this diagram reveal the following ways in which the patterns complement and complete each other in multiple ways to form a pattern language:

- Although each pattern is useful in isolation, the pattern language is even more powerful, because it integrates solutions to particular problems in important technical areas, such as event handling, connection management and

[2] Since it is beyond the scope of this paper to describe each pattern in detail, please see the references [5] for comprehensive coverage of these patterns.

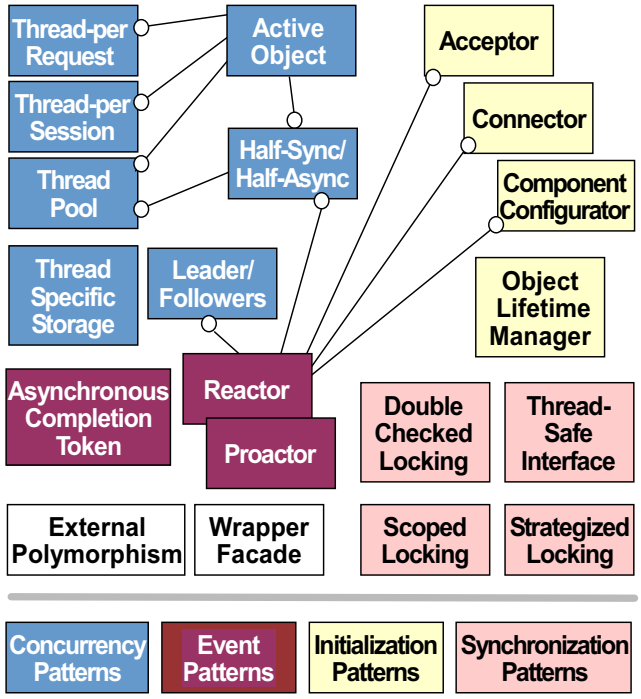

Figure 3: A Pattern Language for Network Programming

service access, concurrency models, and synchronization strategies. Each problem in these areas must be resolved coherently and consistently when developing concurrent and networked applications.

- The pattern language also exposes the interdependencies of these technical areas. For example, when selecting a particular event-handling pattern for a networked application, not all potentially-available concurrency patterns can be applied usefully.
- A pattern language describes an entire solution space for developing a particular type of system, system part, or system aspect – by offering alternative solutions to certain problems. A pattern language can therefore be applied many, many times without ever producing exactly the same architecture, design, or implementation. Yet each individual architecture, design, and implementation follows the same timeless, thus reusable, design knowledge embedded in each pattern language [23].

These three points become clear only when connecting patterns into a pattern language since each pattern in isolation only focuses on itself. In contrast, a pattern language-based design integrates the patterns consistently and synergistically by giving concrete and precise guidance in developing a specific (type of) system or system aspect. In particular, a pattern language conveys what are the key problems to be resolved, in what order should they be tackled, what alternatives exist for resolving a given problem, how are mutual dependencies between the problems handled, and how is each individual problem resolved effectively in the presence of its associated problems.

Many pattern languages exist today. Some support the creation of a specific types of software, such as the pattern languages for networked and concurrent computing [5] and enterprise application architectures [13]. Others pattern languages help to realize a certain part, subsystem, or large component of a software system. For instance, there is a pattern language for server components [12] that describes how to build well-structured server-side components. Yet other languages focus on a particular problem area of relevance in software, such as handling erroneous user input [24] and designing user-friendly interfaces to computer and information systems [25]. There are also pattern languages for good programming style and practice, such as Smalltalk best practice patterns [26], C++ reference accounting [27], and Java exception handling [28]. Section 6 describes other key areas where we expect future pattern languages will be published.

5 Illustrating the Synergies of Middleware, Frameworks, and Patterns

Sections 2–4 present overviews of middleware, frameworks, and patterns, emphasizing their particular contributions to software reuse. This section examines these technologies in terms of their relationships and synergies. To make the discussion tangible, we focus on The ACE ORB (TAO) [29] as a case study to illustrate how patterns and frameworks can help middleware developers build and evolve software by reducing the coupling between its components.

5.1 Overview of CORBA and TAO

CORBA Object Request Brokers (ORBs) [30] allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware. Figure 4 illustrates the components in the CORBA reference model, which collaborate to provide the portability, interoperability, and transparency mentioned above. The client stubs marshal client operations into General Inter-ORB Protocol (GIOP) requests that are transmitted to objects via the standard Internet Inter-ORB Protocol (IIOP)[3] implemented in the ORB Core. The server ORB Core demultiplexes and receives these requests using one or more threads and passes them to the Object Adapter. The Object Adapter demultiplexes the requests to skeletons, which demarshal the requests and dispatch the

[3] IIOP is implemented atop TCP/IP.

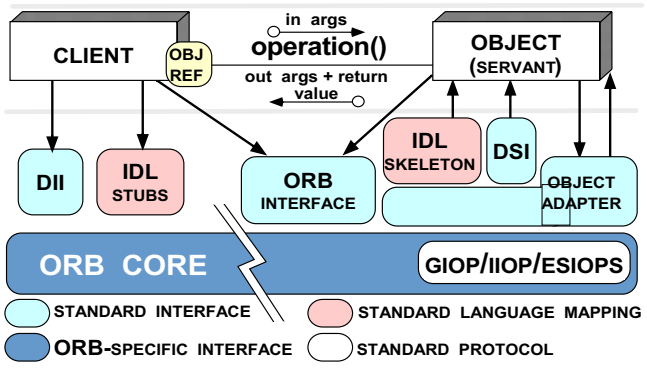

Figure 4: Components in the CORBA Reference Model

appropriate application-level servant method that implements the object's operation.

The ACE ORB (TAO) [29] is a high-performance, real-time implementation of the CORBA 3.0 specification [31] that supports the distribution middleware capabilities shown in Figure 4 and described in Section 2. TAO is targeted for applications with deterministic and statistical QoS requirements, as well as best effort requirements. As its name suggests, TAO is developed using the frameworks and components in the ACE host infrastructure middleware. ACE and TAO are open-source software that has evolved organically over the past decade. They are used in hundreds of production software systems in research labs and commercial projects, including avionics mission computing systems at Boeing, satellite communication systems at Lockheed and Raytheon, telecommunication systems at BBN, Cisco, Lucent, Motorola, Nortel, and Siemens, medical systems at GE, and online trading systems at Automated Trading Desk. Complete source code and documentation for ACE and TAO is available at `www.cs.wustl.edu/~schmidt/TAO/`.

5.2 Applying Patterns and Frameworks to TAO Middleware

The interfaces, protocols, and components shown in Figure 4 illustrate the primary capabilities provided by the CORBA reference architecture. As is often the case with architectural diagrams, however, it is not clear from this figure *how* the architecture behaves or *why* the architecture is designed the way it is. Naturally, a thorough understanding of these issues is essential to develop, configure, optimize, and evolve the reusable middleware effectively. To address these issues, we outline the key patterns and frameworks used by TAO's ORB Core to illustrate the synergies between these two reuse technologies in terms of improving the extensibility, maintainability, and performance of distribution middleware.

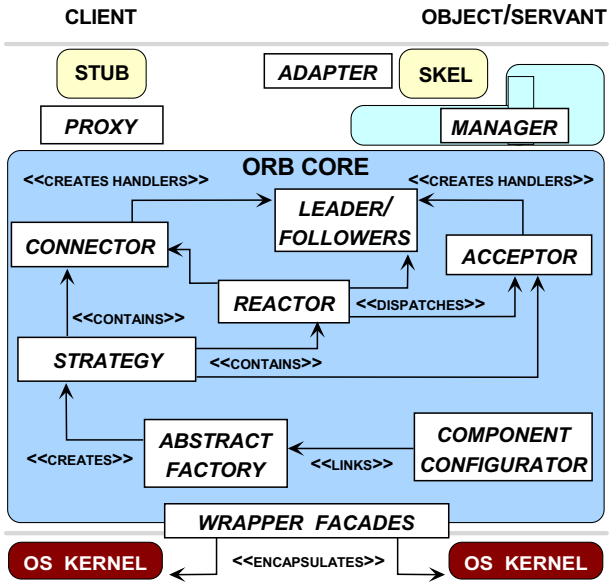

Figure 5: The Patterns Used in TAO

The key patterns used in the TAO ORB are shown in Figure 5 and described below:

- The **Wrapper Facade** pattern [5] simplifies the OS system programming interface by combining multiple related OS mechanisms, such as the socket API or POSIX threads, into cohesive OO abstractions. TAO uses this pattern to enhance its reuse by encapsulating the tedious, non-portable, and non-typesafe low-level OS system functions within C++ classes.
- The **Reactor** pattern [5] allows event-driven applications to react to events originating from a number of disparate sources, such as I/O handles, timers, and signals. TAO uses this pattern to dispatch ORB connection and I/O handlers in response to events that occur in the OS kernel.
- The **Acceptor-Connector** pattern [5] decouples the connection and initialization of cooperating peer services in a networked application from the processing they perform after being connected and initialized. TAO uses this pattern in its ORB Core to actively/passively establish connections and create I/O handlers that exchange GIOP messages independently of the underlying transports.
- The **Leader Followers** pattern [5] provides an efficient concurrency model where multiple threads take turns calling a synchronous event demultiplexer (such as `select()`) on sets of I/O handles to detect, demultiplex, dispatch, and process service requests that occur. TAO uses this pattern to improve its performance and predictability by reducing context switching, synchronization, and data allocation/movement overhead.
- The **Strategy** pattern [3] provides a means to select one of several candidate policies or algorithms and packaging

it with an object. This pattern is the foundation of TAO's extensible software architecture and makes it possible to configure custom ORB strategies for connection management, concurrency, and demultiplexing.

- The **Abstract Factory** pattern [3] provides a single component that builds related objects. TAO uses this pattern to create and consolidate its many strategy objects into a manageable number of abstract factories that can be reconfigured *en masse* into its ORB Core conveniently and consistently. TAO components use these factories to access related strategies without explicitly specifying their subclass name.
- The **Component Configurator** pattern [5] permits dynamic run-time configuration of abstract factories and strategies in an ORB [5]. TAO uses this pattern to dynamically link the abstract factories that produce custom ORB personalities, such as a high-throughput ORB, a predictable real-time ORB, or a small footprint ORB.
- The **Proxy** pattern [4, 3] defines a object (the proxy) that acts as surrogate for a (potentially remote) target object. TAO uses this pattern to perform the (de)marshaling operations defined by CORBA stubs.
- The **Adapter** pattern [3] allows objects to work together that have different interfaces. TAO uses this pattern in its object adapters and skeletons to integrate object implementations (servants) with the reusable ORB code. An object adapter also implements the **Manager** pattern [4] to control the lifecycle of the objects and servants it contains.

Knowing the patterns used in TAO is useful since it enables reuse of abstract design and architecture knowledge. However, reuse of patterns alone is insufficient since it does not directly yield flexible and efficient software. We therefore found it necessary to augment our study of patterns with the creation and use of the frameworks provided by ACE. These frameworks help developers of TAO middleware avoid costly reinvention of common software artifacts by refactoring recurring implementation roles. They also provide direct knowledge transfer by embodying patterns in a powerful toolkit that contains middleware domain experience *and* working code.

The primary ACE frameworks used in TAO's ORB Core are shown in Figure 6. Below, we describe each framework, outline which pattern(s) it implements, and discuss how it enhances reuse.

- The **Reactor** framework implements the Reactor and Leader/Followers patterns to facilitate a concurrency model where events indicating the ability to begin I/O operations. This framework enhances reuse in TAO by automating the detection, demultiplexing, and dispatching of *handlers* in response to I/O, timer, and signal events. The ACE Reactor drives the main event loop

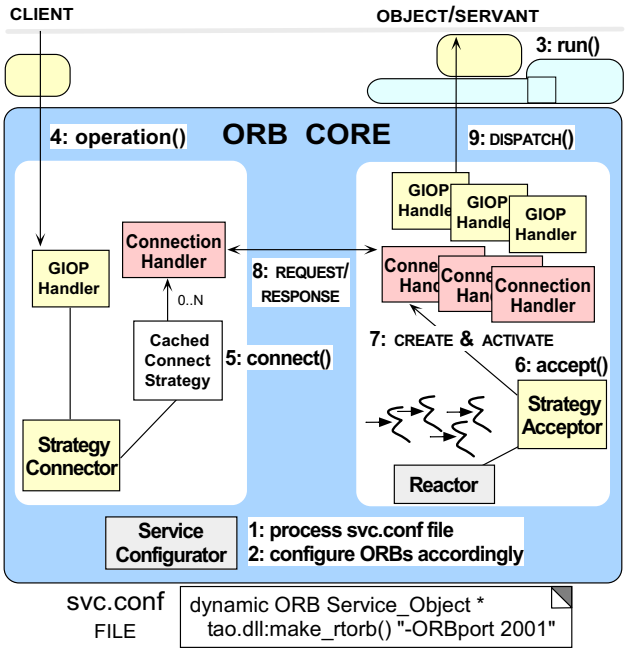

Figure 6: The ACE Frameworks Used in TAO

in TAO, which accepts connections and receives/sends client GIOP requests/responses via a pool of threads.

- The **Acceptor-Connector** framework leverages the Reactor framework and implements the Acceptor-Connector and Strategy patterns to decouple the active and passive initialization roles from application-defined service processing performed by communicating peer services after initialization is complete. This ACE framework enhances reuse by defining (1) *strategy connector* objects that automatically establish connections with servers and initialize handlers that transmit GIOP requests and (2) *strategy acceptor* objects that automatically accept connections from clients and initialize handlers that process GIOP requests from clients.

- The **Service Configurator** framework implements the Component Configurator pattern to support the flexible configuration of ORBs whose strategies can be determined and deployed at the appropriate time in the design cycle, *e.g.*, via the `svc.conf` configuration script at installation time and/or run time. This framework enhances reuse by enabling ORB abstract factories (and the strategies they embody) to be treated as interchangeable building blocks. Software with high availability requirements, such as online transaction processing or real-time industrial process automation, often require these flexible configuration capabilities.

The design of the ACE frameworks presented above is guided by many of the patterns shown in Figures 3 and 5. TAO uses these patterns and the ACE frameworks that reify them to develop an extensible ORB architecture. However, the patterns and frameworks described here are not limited to middleware. In fact, they have been applied in many other application domains, including telecom call processing [32] and switching [8], airplane flight control systems [15], multimedia videoconferencing [33], distributed interactive simulations [34], and enterprise business applications [20].

The patterns described above help to solve many common problems that arise when developing and using frameworks for middleware and applications. In the context of TAO, for example, a deep understanding of these patterns helps to

- **Preserve important design information** for developers who enhance and maintain the TAO middleware. Since scores of developers have worked on TAO over the past decade, this information would be lost if it was not documented, which would increase software entropy and decrease software maintainability and quality. The metrics reported in [35] quantify the extent to which knowledge of patterns helped to reduce the development and maintenance effort for TAO.
- **Guide design choices** for developers who are building new middleware and applications using TAO. The books in which the patterns outlined above appear document the common traps and pitfalls of developing middleware. This information helps TAO developers select suitable architectures, protocols, algorithms, and platform features without wasting time and effort (re)implementing solutions that have been shown to be inefficient or error prone.

5.3 Evaluating the Synergies Between Middleware, Frameworks, and Patterns

The short case study above illustrates how patterns and frameworks are both important techniques to achieve large-scale reuse by capturing successful software development strategies within a particular context, which in this case was distribution and host infrastructure middleware. All three of these technologies help to simplify the development, configuration, and optimization of software by codifying the accumulated expertise of developers who have successfully confronted similar problems before as follows:

- Frameworks codify this expertise in the form of reusable algorithms and component implementations.
- Patterns codify this expertise at a complementary level of reuse – recurring architectural design themes – which may be reused even when the algorithms or component implementations are not directly reusable.
- Middleware codifies this expertise in the form of interfaces and standard components that provide applications with a simpler facade to access the powerful (and complex) internal component structure of frameworks.

For example, as shown in our case study above, TAO middleware developers are more effective since the frameworks in ACE implement the core patterns associated with service access and configuration, event handling, interprocess communication, concurrency, and synchronization.

It is important to recognize that middleware, frameworks, and patterns are highly synergistic concepts, with none subordinate to the other [18]. Patterns can be characterized as more abstract descriptions of frameworks, which are often independent of a particular programming language, operating system, network, or database environment. Patterns have been used to document frameworks and middleware. Sophisticated middleware frameworks are concrete realizations of groups of dozens to hundreds of patterns. A framework also integrates various approaches to problems where there are no *a priori* context-independent and optimal solutions.

6 Future R&D Focus Areas

Over the past decade, R&D on patterns, frameworks, and middleware has focused largely on developing and refining the core concepts and infrastructure that can create the foundations for subsequent efforts [6, 7]. As a result of these advances, we expect the next generation of middleware will be developed using frameworks that consciously embody time-proven patterns. Patterns will also increasingly be used to document the form, content, and best practices of middleware and frameworks. Other key topics and domains that will benefit from the foundational work conducted thus far will include:

- **Distributed real-time and embedded systems**. An increasing number of patterns associated with middleware frameworks for concurrent and networked objects have been documented during the past five years [36, 5, 35]. A key next step is to document the patterns for distributed real-time and embedded (DRE) systems, which extends earlier work to focus on effective strategies and tactics for managing key QoS properties in DRE systems, including network bandwidth and latency, CPU speed, memory access time, and power levels. Since developing high-quality DRE systems is hard and remains somewhat of a "black art," relatively few reusable patterns [37, 38], frameworks [39], and middleware [29] exist for this domain today. We expect an increased focus on DRE systems in the future, however, due to the maturation of reusable object technology, together with the development tools, techniques, and processes that enable it to be applied successfully in the DRE domain.

- **Mobile systems**. Wireless networks are becoming pervasive and embedded devices are become smaller, lighter, and more capable. Thus, mobile systems will soon support many consumer communication and computing needs. Application areas for mobile systems include ubiquitous computing, mobile agents, personal assistants, position-dependent information provision, remote medical diagnostics and teleradiology, and home and office automation. In addition, Internet services, ranging from Web browsing to on-line banking, will be accessed from mobile systems. Mobile systems present many challenges, such as managing low and variable bandwidth and power, adapting to frequent disruptions in connectivity and service quality, diverging protocols, and maintaining cache consistency across disconnected network nodes. We expect that experienced developers of mobile systems will capture their expertise in the form of reusable pattern, frameworks, and middleware to help meet the growing demand for quality software in this area.
- **Adaptive QoS for COTS systems**. Distributed applications, such as streaming video, Internet telephony, and large-scale interactive simulation systems, have increasingly stringent QoS. To reduce development cycle-time and cost, these applications are increasingly being developed using multiple layers of COTS hardware, operating systems, and middleware components, such as those presented in Section 2. Historically, however, it has been hard to configure COTS-based systems that can simultaneously satisfy multiple QoS properties, such as security, timeliness, and fault tolerance [40]. As developers and integrators continue to master the complexities of providing end-to-end QoS guarantees, it is essential that they document the successful patterns and reify them in the form of reusable adaptive and reflective middleware frameworks [41] to help others configure, monitor, and control COTS-based distributed systems that possess a range of interdependent QoS properties.

7 Concluding Remarks

Application software has historically been developed largely from scratch. This development process has been applied many times in many companies, by many projects in parallel. Even worse, it has been applied by the same teams in a series of projects. Regrettably, this continuous rediscovery and reinvention of core concepts and code has kept costs unnecessarily high throughout the software development life cycle. This problem is exacerbated by the inherent diversity of today's hardware, operating systems, compilers, and communication platforms, which keep shifting the foundations of application software development.

In today's competitive, time-to-market-driven environments, it is increasingly infeasible to develop custom solutions manually from scratch. Such solutions are hard to customize and tune, because so much effort is spent just trying to make the software operational. Moreover, as requirements change over time, evolving custom software solutions becomes prohibitively expensive. End-users expect–or at least desire–software to be affordable, robust, efficient, and agile, which is hard to achieve without the solid architectural underpinnings achievable via systematic reuse.

The past decade has yielded significant progress in reuse of software in the form of the maturation of standard middleware, the documentation of patterns, and the development and adoption of frameworks. These software technologies provide the following general types of improvement for developing and evolving application software:

1. **Open standards**, which provide a portable and interoperable set of software artifacts, such as interoperable security, layered distributed resource management, and fault tolerance services. An increasingly important role is being played by open and/or standard COTS middleware frameworks that can be purchased or acquired via open-source means. COTS middleware frameworks are particularly important for organizations facing time-to-market pressures and limited software development resources.
2. **Strategic focus**, which elevates application developer focus beyond a preoccupation with low-level OS APIs. For example, the standard middleware artifacts outlined above help to direct the focus of developers toward higher-level software application architecture and design concerns. Without needing to worry as much about low-level details, developers can focus on more strategic, application-oriented concerns.
3. **Design reuse**, patterns are essential to guiding developers through the steps necessary to ensure successful creation and deployment of complex software systems. In particular, patterns enable developers to reuse higher-level software application designs, such as publisher/subscriber architectures, micro-kernels, and brokers [4]. These design artifacts represent some of the key strategic aspects of complex software. If they are understood and applied properly, the impact of many vexing complexities can be alleviated greatly.
4. **Implementation reuse**, which amortizes software lifecycle effort by leveraging previous development expertise and reifying implementations of key patterns [5, 3] into reusable middleware frameworks. In the future, most applications will be assembled by integrating and scripting domain-specific and common "pluggable" middleware service components, rather than being programmed entirely from scratch.

Despite their natural synergies, however, middleware, frameworks, and patterns are not silver bullets. They cannot, for example, absolve developers from responsibility for solving all complex concurrent and networked software analysis, design, implementation, validation, and optimization problems. Ultimately there is no substitute for human creativity, experience, discipline, diligence, and judgement. When used together properly, however, the technologies described in this paper help alleviate many inherent and accidental software complexities.

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston: Addison-Wesley, 2002.

[2] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2002.

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

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996.

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

[6] M. Fayad, R. Johnson, and D. C. Schmidt, eds., *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. New York: Wiley & Sons, 1999.

[7] M. Fayad, R. Johnson, and D. C. Schmidt, eds., *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. New York: Wiley & Sons, 1999.

[8] H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," in *Proceedings of OOPSLA '95*, (Austin, TX), ACM, Oct. 1995.

[9] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Boston: Addison-Wesley, 2002.

[10] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Reading, Massachusetts: Addison-Wesley, 2002.

[11] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[12] M. Volter, A. Schmid, and E.Wolff, *Server Component Patterns – Component Infrastructures illustrated with EJB*. New York: Wiley & Sons, 2002.

[13] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, *Patterns of Enterprise Application Architecture*. Reading, Massachusetts: Addison-Wesley, 2002.

[14] F. Buschmann, A. Geisler, T. Heimke, and C. Schuderer, "Framework-Based Software Architectures for Process Automation Systems," in *Proceedings of the 9th IFAC Symposium on Automation in Mining, Mineral, and Metal Processing*, (Cologne, Germany), 1998.

[15] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[16] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[17] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June/July 1988.

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

[19] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," *IEEE Software*, vol. 15, November/December 1998.

[20] M. Fayad and R. Johnson, eds., *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. New York: Wiley & Sons, 1999.

[21] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. New York, NY: Oxford University Press, 1977.

[22] J. O. Coplien and D. C. Schmidt, eds., *Pattern Languages of Program Design*. Reading, Massachusetts: Addison-Wesley, 1995.

[23] C. Alexander, *The Timeless Way of Building*. New York, NY: Oxford University Press, 1979.

[24] W. Cunningham, "The CHECKS Pattern Language of Information Integrity," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, Massachusetts: Addison-Wesley, 1995.

[25] J. Borchers, *"A Pattern Approach To Interaction Design"*. New York: Wiley & Sons, 2001.

[26] K. Beck, *Smalltalk Best Practice Patterns*. Englewood Cliffs, NJ: Prentice-Hall, 1997.

[27] K. Henney, "C++ Patterns - Reference Accounting," in *Proceedings of the EuroPLoP 2002 conference*, (Irsee, Germany), July 2002.

[28] A. Haase, "Java Idioms: Exception Handling," in *Proceedings of the EuroPLoP 2003 conference*, (Irsee, Germany), July 2003.

[29] 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.

[30] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley, 1999.

[31] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0 ed., June 2002.

[32] G. Meszaros, "A Pattern Language for Improving the Capacity of Reactive Systems," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, Massachusetts: Addison-Wesley, 1996.

[33] D. C. Schmidt, V. Kachroo, Y. Krishnamurthy, and F. Kuhns, "Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications," *IEEE Communications Magazine*, vol. 38, pp. 112–123, Oct. 2000.

[34] R. Noseworthy, "IKE 2 – Implementing the Stateful Distributed Object Paradigm ," in *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, (Washington, DC), IEEE, Apr. 2002.

[35] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, Apr. 1999.

[36] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Boston: Addison-Wesley, 2000.

[37] D. Lea and J. Marlowe, "PSL: Protocols and Pragmatics for Open Systems," in *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, Aug. 1995.

[38] J. Noble and C. Weir, *Small Memory Software: Patterns for Systems with Limited Memory*. Boston: Addison-Wesley, 2001.

[39] D. C. Schmidt, "R&D Advances in Middleware for Distributed, Real-time, and Embedded Systems," *Communications of the ACM special issue on Middleware*, vol. 45, pp. 43–48, June 2002.

[40] 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, pp. 1–20, 1997.

[41] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications ACM*, vol. 45, pp. 33–38, June 2002.