Patterns, Frameworks, & Middleware: Their Synergistic Relationships

Frontiers of Software Practice 2003

Douglas C. Schmidt
d.schmidt@vanderbilt.edu

Professor of EECS
Vanderbilt University
Nashville, Tennessee
Information technology is being commoditized
• i.e., hardware & software are getting cheaper, faster, & (generally) better at a fairly predictable rate

These advances stem largely from standard hardware & software APIs & protocols, e.g.:
• Intel x86 & Power PC chipsets
• TCP/IP, GSM, Link16
• POSIX, Windows, & VMs
• Middleware & component models
• Quality of service (QoS) aspects
Growing acceptance of a network-centric component paradigm
• i.e., distributed applications with a range of QoS needs are constructed by integrating components & frameworks via various communication mechanisms
Component middleware is maturing & becoming pervasive

- **Components** encapsulate application “business” logic
- Components interact via **ports**
  - **Provided interfaces**, e.g., facets
  - **Required connection points**, e.g., receptacles
- **Event sinks & sources**
- **Attributes**

- **Containers** provide execution environment for components with common operating requirements
- Components/containers can also
  - Communicate via a **middleware bus** and
  - Reuse **common middleware services**
Historically, mission-critical apps were built directly atop hardware & OS
- Tedious, error-prone, & costly over lifecycles

There are layers of middleware, just like there are layers of networking protocols

Standards-based COTS middleware helps:
- Control end-to-end resources & QoS
- Leverage hardware & software technology advances
- Evolve to new environments & requirements
- Provide a wide array of reuseable, off-the-shelf developer-oriented services
• Operating systems & protocols provide mechanisms to manage endsystem resources, e.g.,
  • CPU scheduling & dispatching
  • Virtual memory management
  • Secondary storage, persistence, & file systems
  • Local & remove interprocess communication (IPC)

• OS examples
  • UNIX/Linux, Windows, VxWorks, QNX, etc.

• Protocol examples
  • TCP, UDP, IP, SCTP, RTP, etc.
Host Infrastructure Middleware

- Host infrastructure middleware encapsulates & enhances native OS mechanisms to create reusable network programming components
  - These components abstract away many tedious & error-prone aspects of low-level OS APIs

- Examples
  - Java Virtual Machine (JVM), Common Language Runtime (CLR), ADAPTIVE Communication Environment (ACE)

www.rtj.org

www.cs.wustl.edu/~schmidt/ACE.html
Distribution Middleware

- **Distribution middleware** defines higher-level distributed programming models whose reusable APIs & components automate & extend native OS capabilities.

- **Examples**
  - OMG CORBA, Sun’s Remote Method Invocation (RMI), Microsoft’s Distributed Component Object Model (DCOM)

- Distribution middleware avoids hard-coding client & server application dependencies on object location, language, OS, protocols, & hardware.
Common Middleware Services

- **Common middleware services** augment distribution middleware by defining higher-level domain-independent services that focus on programming “business logic”
- **Examples**
  - CORBA Component Model & Object Services, Sun’s J2EE, Microsoft’s .NET
  - Common middleware services support many recurring distributed system capabilities, e.g.,
    - Transactional behavior
    - Authentication & authorization,
    - Database connection pooling & concurrency control
    - Active replication
    - Dynamic resource management
**Domain-Specific Middleware**

*Domain-specific middleware services* are tailored to the requirements of particular domains, such as telecom, e-commerce, health care, process automation, or aerospace.

**Examples**

- **Siemens MED Syngo**
  - Common software platform for distributed electronic medical systems
  - Used by all ~13 Siemens MED business units worldwide

- **Boeing Bold Stroke**
  - Common software platform for Boeing avionics mission computing systems

**Modalities**
- e.g., MRI, CT, CR, Ultrasound, etc.
Why We are Succeeding

The past decade has yielded significant progress in QoS-enabled middleware, stemming in large part from the following trends:

- Years of iteration, refinement, & successful use
- The maturation of middleware standards
- The maturation of component middleware frameworks & patterns

- NET, J2EE, CCM
- Real-time CORBA
- Real-time Java
- SOAP & Web Services

- CORBA Component Model (CCM)
- Component Models (EJB)
- Real-time CORBA
- CORBA & DCOM
- DCE
- Micro-kernels
- RPC
- ARPAnet

1970 Year 2005
Overview of Patterns

- Present solutions to common software problems arising within a certain context
- Help resolve key software design forces
  - Flexibility
  - Extensibility
  - Dependability
  - Predictability
  - Scalability
  - Efficiency

- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs
- Generally codify expert knowledge of design strategies, constraints & “best practices”

**The Proxy Pattern**

```
Client

AbstractService

Proxy

Service

1

1
```

*Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Overview of Pattern Languages

Motivation

- Individual patterns & pattern catalogs are insufficient
- Software modeling methods & tools largely just illustrate \textit{how} – not \textit{why} – systems are designed

Benefits of Pattern Languages

- Define a \textit{vocabulary} for talking about software development problems
- Provide a \textit{process} for the orderly resolution of these problems
- Help to generate & reuse software \textit{architectures}
## Taxonomy of Patterns & Idioms

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Idioms</strong></td>
<td>Restricted to a particular language, system, or tool</td>
<td>Scoped locking</td>
</tr>
<tr>
<td><strong>Design patterns</strong></td>
<td>Capture the static &amp; dynamic roles &amp; relationships in solutions that occur repeatedly</td>
<td>Active Object, Bridge, Proxy, Wrapper Façade, &amp; Visitor</td>
</tr>
<tr>
<td><strong>Architectural patterns</strong></td>
<td>Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, &amp; include the rules and guidelines for organizing the relationships between them</td>
<td>Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, &amp; Reactor</td>
</tr>
<tr>
<td><strong>Optimization principle patterns</strong></td>
<td>Document rules for avoiding common design &amp; implementation mistakes that degrade performance</td>
<td>Optimize for common case, pass information between layers</td>
</tr>
</tbody>
</table>
Example: Boeing Bold Stroke

- Avionics mission computing product-line architecture for Boeing military aircraft, e.g., F-18 E/F, 15E, Harrier, UCAV
- DRE system with 100+ developers, 3,000+ software components, 3-5 million lines of C++ code
- Based on COTS hardware, networks, operating systems, & middleware
- Used as Open Experimentation Platform (OEP) for DARPA IXO PCES, MoBIES, SEC, MICA programs
Example: Boeing Bold Stroke

COTS & Standards-based Middleware Infrastructure, OS, Network, & Hardware Platform

- Real-time CORBA middleware services
- VxWorks operating system
- VME, 1553, & Link16
- PowerPC
Example: Boeing Bold Stroke

Reusable Object-Oriented Application Domain-specific Middleware Framework
- Configurable to variable infrastructure configurations
- Supports systematic reuse of mission computing functionality
**Example: Boeing Bold Stroke**

**Product Line Component Model**
- Configurable for product-specific functionality & execution environment
- Single component development policies
- Standard component packaging mechanisms
Example: Boeing Bold Stroke

Component Integration Model
- Configurable for product-specific component assembly & deployment environments
- Model-based component integration policies
Legacy Avionics Architectures

Key System Characteristics
• Hard & soft real-time deadlines
  • ~20-40 Hz
• Low latency & jitter between boards
  • ~100 usecs
• Periodic & aperiodic processing
• Complex dependencies
• Continuous platform upgrades

Avionics Mission Computing Functions
• Weapons targeting systems (WTS)
• Airframe & navigation (Nav)
• Sensor control (GPS, IFF, FLIR)
• Heads-up display (HUD)
• Auto-pilot (AP)

4: Mission functions perform avionics operations
3: Sensor proxies process data & pass to missions functions
2: I/O via interrupts
1: Sensors generate data
Legacy Avionics Architectures

Key System Characteristics
• Hard & soft real-time deadlines
  • ~20-40 Hz
• Low latency & jitter between boards
  • ~100 usecs
• Periodic & aperiodic processing
• Complex dependencies
• Continuous platform upgrades

Limitations with Legacy Avionics Architectures
• Stovepiped
• Proprietary
• Expensive
• Vulnerable
  • Tightly coupled
• Hard to schedule
• Brittle & non-adaptive
### Decoupling Avionics Components

**Context**
- I/O driven DRE application
- Complex dependencies
- Real-time constraints

**Problems**
- Tightly coupled components
- Hard to schedule
- Expensive to evolve

**Solution**
- Apply the **Publisher-Subscriber** architectural pattern to distribute periodic, I/O-driven data from a single point of source to a collection of consumers

<table>
<thead>
<tr>
<th><strong>Structure</strong></th>
<th><strong>Publisher</strong></th>
<th><strong>Event Channel</strong></th>
<th><strong>Subscriber</strong></th>
<th><strong>Filter</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>produce</td>
<td>attachPublisher, detachPublisher, attachSubscriber, detachSubscriber, pushEvent</td>
<td>creates, receives</td>
<td>consume</td>
<td>filterEvent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Dynamics</strong></th>
<th><strong>: Publisher</strong></th>
<th><strong>: Event Channel</strong></th>
<th><strong>: Subscriber</strong></th>
<th><strong>: Event</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>produce</td>
<td>attachSubscriber</td>
<td>pushEvent event, detachSubscriber</td>
<td>consume</td>
<td></td>
</tr>
</tbody>
</table>
Applying the Publisher-Subscriber Pattern to Bold Stroke

Bold Stroke uses the **Publisher-Subscriber** pattern to decouple sensor processing from mission computing operations:

- Anonymous publisher & subscriber relationships
- Group communication
- Asynchrony

Considerations for implementing the **Publisher-Subscriber** pattern for mission computing applications include:

- **Event notification model**
  - Push control vs. pull data interactions
- **Scheduling & synchronization strategies**
  - e.g., priority-based dispatching & preemption
- **Event dependency management**
  - e.g., filtering & correlation mechanisms
Ensuring Platform-neutral & Network-transparent Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Mission computing requires remote IPC</td>
<td>• Applications need capabilities to:</td>
<td>• Apply the <strong>Broker</strong> architectural pattern to provide platform-neutral communication between mission computing boards</td>
</tr>
<tr>
<td>• Stringent DRE requirements</td>
<td>• Support remote communication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Provide location transparency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Handle faults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Manage end-to-end QoS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Encapsulate low-level system details</td>
<td></td>
</tr>
</tbody>
</table>

**Context**
- Mission computing requires remote IPC
- Stringent DRE requirements

**Problems**
- Applications need capabilities to:
  - Support remote communication
  - Provide location transparency
  - Handle faults
  - Manage end-to-end QoS
  - Encapsulate low-level system details

**Solution**
- Apply the **Broker** architectural pattern to provide platform-neutral communication between mission computing boards

**Structure**

**Client Proxy**
- marshal
- unmarshal
- receive_result
- service_p

**Server Proxy**
- marshal
- unmarshal
- dispatch
- receive_request

**Client**
- call_service_p
- start_task

**Broker**
- main_loop
- srv_registration
- srv_lookup
- xmit_message
- manage_QoS

**Server**
- start_up
- main_loop
- service_i
Ensuring Platform-neutral & Network-transparent Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Mission computing requires remote IPC | Applications need capabilities to:  
- Support remote communication  
- Provide location transparency  
- Handle faults  
- Manage end-to-end QoS  
- Encapsulate low-level system details | Apply the **Broker** architectural pattern to provide platform-neutral communication between mission computing boards |

**Context**:  
- Broker: Client Proxy  
- Client: Server Proxy  
- Server: Client |

**Dynamics**  
- Operation (params)  
- Connect  
- Marshal  
- Send request  
- Receive reply  
- Marshal  
- Result  
- Start up
Applying the Broker Pattern to Bold Stroke

Bold Stroke uses the **Broker** pattern to shield distributed applications from environment heterogeneity, e.g.,
- Programming languages
- Operating systems
- Networking protocols
- Hardware

A key consideration for implementing the **Broker** pattern for mission computing applications is **QoS** support
- e.g., latency, jitter, priority preservation, dependability, security, etc.

**Caveat**

*These patterns are very useful, but having to implement them from scratch is tedious & error-prone!!!*
Overview of Frameworks

Framework Characteristics

• Frameworks exhibit “inversion of control” at runtime via callbacks

• Frameworks provide integrated domain-specific structures & functionality

• Frameworks are “semi-complete” applications

Application-specific functionality

Mission Computing

E-commerce

Networking

Database

GUI

Scientific Visualization
Comparing Class Libraries, Frameworks, & Components

**Class Library Architecture**

A class is a unit of abstraction & implementation in an OO programming language.

**Framework Architecture**

A framework is an integrated set of classes that collaborate to produce a reusable architecture for a family of applications.

**Component Architecture**

A component is an encapsulation unit with one or more interfaces that provide clients with access to its services.

<table>
<thead>
<tr>
<th>Class Libraries</th>
<th>Frameworks</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-level</td>
<td>Meso-level</td>
<td>Macro-level</td>
</tr>
<tr>
<td>Stand-alone</td>
<td>“Semi-complete”</td>
<td>Stand-alone</td>
</tr>
<tr>
<td>language</td>
<td>applications</td>
<td>composition</td>
</tr>
<tr>
<td>entities</td>
<td></td>
<td>entities</td>
</tr>
<tr>
<td>Domain-independent</td>
<td>Domain-specific</td>
<td>Domain-specific or Domain-independent</td>
</tr>
<tr>
<td>Borrow caller’s thread</td>
<td>Inversion of control</td>
<td>Borrow caller’s thread</td>
</tr>
</tbody>
</table>

**Middleware Bus**

- Naming
- Events
- Logging
- Locking
- Application-specific functionality
- Middleware Bus
- GUI
- DATABASE
- CALLBACKS
- APPLICATION-SPECIFIC FUNCTIONALITY
- INVOKES
- Files
- Locks
- Strings
- ADTs
Observations
• Frameworks are powerful, but hard to develop & use effectively by application developers
  • It’s often better to use & customize COTS frameworks than to develop in-house frameworks
• Components are easier for application developers to use, but aren’t as powerful or flexible as frameworks

Successful projects are therefore often organized using the “funnel” model
Overview of the ACE Frameworks

Features
- Open-source
- 6+ integrated frameworks
- 250,000+ lines of C++
- 40+ person-years of effort
- Ported to Windows, UNIX, & real-time operating systems (e.g., VxWorks, pSoS, LynxOS, Chorus, QNX)
- Large user community

www.cs.wustl.edu/~schmidt/ACE.html
The POSA2 Pattern Language

Pattern Benefits

• Preserve crucial design information used by applications & middleware frameworks & components
• Facilitate reuse of proven software designs & architectures
• Guide design choices for application developers
Implementing the Broker Pattern for Bold Stroke Avionics

- **CORBA** is a distribution middleware standard
- **Real-time CORBA** adds QoS to classic CORBA to control:
  1. **Processor Resources**
  2. **Communication Resources**
  3. **Memory Resources**

- These capabilities address some (but by no means all) important DRE application development & QoS-enforcement challenges
Applying Patterns & Frameworks to Middleware:

The ACE ORB (TAO)

• TAO is an open-source version of Real-time CORBA

• TAO Synopsis
  • > 1,000,000 SLOC
  • 80+ person years of effort

• Pioneered R&D on DRE middleware design, patterns, frameworks, & optimizations

• TAO is basis for many middleware R&D efforts

• Example of good synergy between researchers & practitioners

www.cs.wustl.edu/~schmidt/TAO.html
Key Patterns Used in TAO


- **Wrapper facades** enhance portability
- **Proxies & adapters** simplify client & server applications, respectively
- **Component Configurator** dynamically configures **Factories**
- **Factories** produce **Strategies**
- **Strategies** implement interchangeable policies
- Concurrency strategies use **Reactor & Leader/Followers**
- **Acceptor-Connector** decouples connection management from request processing
- **Managers** optimize request demultiplexing
### Enhancing ORB Flexibility w/the Strategy Pattern

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-domain resuable middleware framework</td>
<td>Flexible ORBs must support multiple event &amp; request demuxing, scheduling, (de)marshaling, connection mgmt, request transfer, &amp; concurrency policies</td>
<td>Apply the <strong>Strategy</strong> pattern to factory out similarity amongst alternative ORB algorithms &amp; policies</td>
</tr>
</tbody>
</table>

#### Solution

- **Hook for marshaling strategy**
- **Hook for the request demuxing strategy**
- **Hook for the event demuxing strategy**
- **Hook for the connection management strategy**
- **Hook for the concurrency strategy**
- **Hook for the underlying transport strategy**

![Diagram of ORB components and hooks](image_url)
## Consolidating Strategies with the Abstract Factory Pattern

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| • A heavily strategized framework or application | • Aggressive use of Strategy pattern creates a configuration nightmare  
  • Managing many individual strategies is hard  
  • It’s hard to ensure that groups of semantically compatible strategies are configured | • Apply the Abstract Factory pattern to consolidate multiple ORB strategies into semantically compatible configurations |

### Concrete factories create groups of strategies
## Dynamically Configuring Factories w/the Component Configurator Pattern

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Resource constrained & highly dynamic environments | Prematurely committing to a particular ORB configuration is inflexible & inefficient  
  - Certain decisions can’t be made until runtime  
  - Forcing users to pay for components that don’t use is undesirable | Apply the **Component Configurator** pattern to assemble the desired ORB factories (& thus strategies) dynamically |

### Solution

**Context**
- ORB strategies are decoupled from when the strategy implementations are configured into an ORB
- This pattern can reduce the memory footprint of an ORB

```plaintext
svc.conf
FILE
 dynamic ORB Service_Object *
  avionics_orb:make_orb() "-ORBport 2001"
```
ACE Frameworks Used in TAO

• Reactor drives the ORB event loop
  - Implements the Reactor & Leader/Followers patterns

• Acceptor-Connector decouples passive/active connection roles from GIOP request processing
  - Implements the Acceptor-Connector & Strategy patterns

• Service Configurator dynamically configures ORB strategies
  - Implements the Component Configurator & Abstract Factory patterns
Summary of Pattern, Framework, & Middleware Synergies

The technologies codify expertise of experienced researchers & developers

- Frameworks codify expertise in the form of reusable algorithms, component implementations, & extensible architectures
- Patterns codify expertise in the form of reusable architecture design themes & styles, which can be reused even when algorithms, components implementations, or frameworks cannot
- Middleware codifies expertise in the form of standard interfaces & components that provide applications with a simpler façade to access the powerful (& complex) capabilities of frameworks

There are now powerful feedback loops advancing these technologies
• There is a limit to how much application functionality can be factored into broadly reusable standard COTS middleware

• Middleware has become extremely complicated to use, configure, & provision statically & dynamically

• There are now (& will always be) multiple middleware technologies to choose from
The Road Ahead (2/2)

Solution approach: Integrate model-based software technologies with QoS-enabled component middleware

- e.g., standard technologies are emerging that can:
  1. Model
  2. Analyze
  3. Synthesize &
  4. Provision

multiple layers of QoS-enabled middleware

- These technologies are guided by patterns & implemented by component frameworks
Concluding Remarks

R&D Synergies

• Researchers & developers of distributed systems face common challenges, e.g.:
  • connection management
  • service initialization
  • error handling
  • flow & congestion control
  • event demultiplexing
  • distribution
  • concurrency & synchronization
  • fault tolerance
  • scheduling &
  • persistence

• Pattern languages, frameworks, & component middleware work together to help resolve these challenges

Key open R&D challenges include:
  • Layered QoS specification & enforcement
  • Separating policies & mechanisms across layers
  • Time/space optimizations for middleware & apps

• Multi-level global resource mgmt. & optimization
  • High confidence
  • Stable & robust adaptive systems

• Prior R&D efforts have address some, but by no means all, of the challenging DOC middleware research topics
• Patterns & frameworks for concurrent & networked objects
  • www.cs.wustl.edu/~schmidt/POSA/
  • www.cs.wustl.edu/~schmidt/ACE/
• ACE & TAO open-source middleware
  • www.cs.wustl.edu/~schmidt/ACE.html
  • www.cs.wustl.edu/~schmidt/TAO.html
• Research papers
  • www.cs.wustl.edu/~schmidt/research.html
• Tutorial on patterns, frameworks, & middleware
  • UCLA extension, July 9-11, 2003
  • www.cs.wustl.edu/~schmidt/UCLA.html
• Conferences on patterns, frameworks, & middleware
  • DOA, ECOOP, ICDCS, ICSE, Middleware, OOPSLA, PLoP(s), RTAS,