Context Object
A Design Pattern for Efficient Information Sharing across Multiple System Layers

Arvind S. Krishna, Douglas C. Schmidt
Electrical Engineering & Computer Science
Vanderbilt University, Nashville, TN
{arvindk, schmidt}@dre.vanderbilt.edu

Michael Stal
Siemens AG Corporate Technology
Munich, Germany
michael.stal@siemens.com

Abstract

Software systems with a layered architecture, such as middleware, need to propagate/share information across the different system layers. In middleware for example, efficient information sharing across different middleware layers enables timely processing of client requests; a critical middleware functionality This paper presents the Context Object pattern that allows efficient processing of requests by propagating context information between different middleware layers. Using this pattern, a layer/session propagates per-request information required by the next layer/session in a context object, which eliminates the need for (1) per-request state within each layer and (2) locking/synchronization to access per-request information across different layers. This pattern is used in CORBA Object Request Brokers (ORBs) for request processing.

1 Intent

This pattern provides an efficient, and application-transparent way of sharing information between different layers in a software system.

2 Example

Many applications use middleware to shield them from system-specific issues, such as communication protocols, concurrency strategies, (de)multiplexing strategies, or (de)marshaling mechanisms. This transparency helps developers focus on application needs, rather than wrestling with low-level distribution details explicitly and manually. To provide the different middleware functionalities, middleware implementers apply the Layers Pattern [POSA1] to group different responsibilities detailed above into groups of tasks at different levels of abstraction. Figure 1 illustrates the different layers in a CORBA based middleware implementations, including the I/O layer (that manages request/response (de)multiplexing), ORB Core layer (that manages message parsing and dispatching), Object Adapter layer (that manages object lifecycle) and Application layer (that hosts application defined entities called servants). Middleware functionalities, such as request/response processing, however, crosscut the different layers.

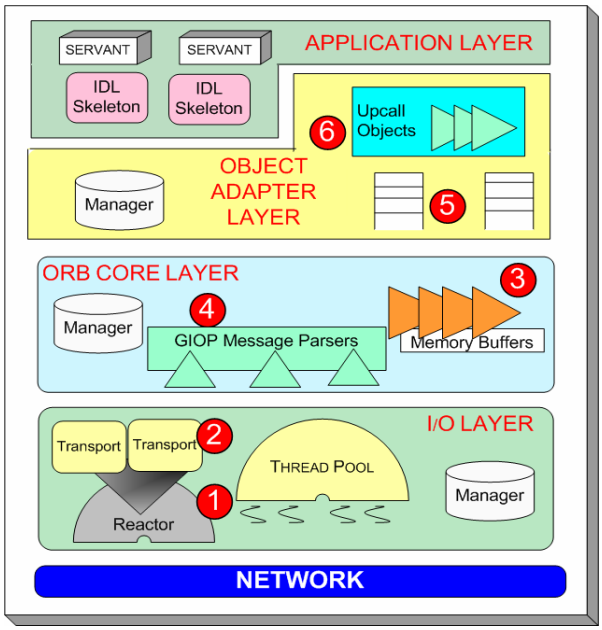


Figure 1. Layered CORBA Request Processing

In our example, as shown in Figure 1 (and discussed in Appendix A), request processing steps 1-2 are carried out in the I/O layer, steps 3-5 are carried out in the ORB core layer, and step 6 is executed in the object adapter layer. In addition, each layer requires some information processed by a layer directly beneath it. For example, the name of the operation demarshaled in the I/O layer is required in the object adapter layer to perform the upcall. Similarly, a message parser associated in the ORB core layer is required in the object adapter layer to create the appropriate reply. *Context information* (*e.g.*, the transport, message parsers and memory buffers associated with each request) must therefore be maintained at each layer and across different layers.

One way to associate and maintain context information at different layers is to create a *manager* [Sommerlad:98] at each layer (as shown in Figure 1) and store the state created within a layer with each request. In our example, a manager in the I/O layer can be used to associate the transport with the request. The object adapter layer can then query this manager to obtain the transport object associated with the request to send the reply. This approach does not

scale well, however, since it (1) forces the ORB to maintain a manager for every objected created as a part of request processing, (2) incurs lookup overhead since information is distributed in different managers, and (3) requires synchronization between the server threads in multi-threaded ORB configurations.

A manager-per-layer model also necessitates removing the state associated in each store after processing a request, which increases the overhead of processing requests. These factors therefore motivate the use of alternative techniques that maximize the efficiency of processing requests, irrespective of the type of request, services associated, and the context information.

3 Context

Software systems that provide rich and configurable set of functionalities resulting in multiple layers.

4 Problem

Communication middleware provides developers with a powerful and rich set of services for building applications. These services often require context information to track relationships between clients and servers. For example, in QoS-enabled middleware implementations, such as Real-time CORBA [RT-CORBA:00], priorities can be conveyed with requests so that middleware implementations can adapt their processing according to these priorities, *i.e.*, by assigning the incoming request to a thread with the appropriate priority. Since this context information often cross-cuts normal request processing at each layer, developers need to determine *how context information can be passed and maintained efficiently* since using *ad hoc* context management mechanisms for each service or middleware layer yields middleware implementations that are hard to maintain, change, or evolve.

Designing middleware implementations that pass context information across middleware layers efficiently is hard since the following forces must be resolved:

- Context information depends on the type of requests and associated QoS policies. For example, the service context information in the request depends on the services associated with the ORB. Similarly, a CORBA `LocateRequest` message (used to determine if the server is capable of handling requests on an object reference) has a different format than a normal CORBA request.
- The specifications for features and options for middleware can be large. For example, new CORBA specifications (such as Deployment and Configuration of Components [DnC:03]) has been recently added by the OMG. Similarly, research efforts are examining integrating fault-tolerance with real-time CORBA [Gokhale:02].
- Explicit (de)allocation of resources along the critical request response path in a middleware implementation should be minimized.

In addressing the previous forces care must be taken to provide a solution that:

- Presents an architecture that can be extended easily, *e.g.*, adding a new protocol may require enhancements to the I/O layer or the ORB core or Object Adapter layers, but should not break the interface between the layers nor increase overhead for other protocols.
- Is transparent to the application, *i.e.*, does not expose any API.

The problem is therefore how (1) context information required for processing different middleware functionalities (such as different services or different message types) can be propagated efficiently within middleware and (2) middleware developers can minimize/eliminate changes to the critical processing path, while adding support for new middleware functionality.

5 Solution

Associate state (such as remote address information, socket information, and the right message parsers) required to process a request with the request. For every request that a client invokes on a service hosted in a server, create a generic *context object* to store all the state necessary to process the request. As the request is processed at each layer, add/remove the state required/redundant for further request processing to the context object. The server uses information in the context object to process the request and send the response to the client. Likewise, the client uses the context object to store request-specific information (such as request priority) and map the response from the server to the right application logic. To minimize memory allocations, recycle the context object after processing the request.

6 Structure

Figure 2 shows the structure of all the participants and their relationships in the context object pattern.

7 Participants

The following are the participants in the Context Object pattern.

- **Layer1:** Create the context object, initialize it, and add context information to the object, and upcall the next layer passing on the context object.
  → In our example the I/O layer creates the context object and adds context information, such as the socket, transport, and buffers associated with the client connection.

Class: Layer 1	Collaborator
Responsibility: • Creates Context Object • Adds context information • Upcalls next Layer passing Context information	• Layer 2 • Context Object Factory

- **LayerJ (where J=2..n):** Receive the Context Object from the layer below it, add context information, and pass it on to the next layer, *i.e*, (J+1). If J is the final layer, complete the request processing and send the reply to the client. The call stack unwinds and Layer 1 can reuse the request for the next request processing cycle.
  → In our example, the ORB layer (layer 2) uses the service context information in the context object created by the I/O layer (layer 1) to take suitable action, such as setting the right priority information and passing it to the object adapter layer (layer 3), which upcalls the application layer (layer 4) and sends the response using the socket information in the context object.

Class: Layer J	Collaborator
Responsibility: • Adds and uses service context information in the context object • Pass it to the next layer if not the last layer	• Layer J+1 • Layer J-1

- **ContextObjectFactory:** Define an interface and a factory method that creates a context object.
  → In our example, the ORB defines a factory class with a `create()` operation defined. The I/O layer uses this factory to create a context object. The factory can also use the recycle method () to cache/recycle context objects.

Class: Context_Object_Factory	Collaborator
Responsibility: • Provides an interface to create different context objects	• Layer 1

Class: Context Object	Collaborator
Responsibility: • Provides accessor and mutator operations to add and retrieve context information	• Layer 1

- **ContextObject:** Define accessor and mutator operations to get/set context information.
  → In our example, the context object has operations `get_end_point_info ()` *to* get the IP address

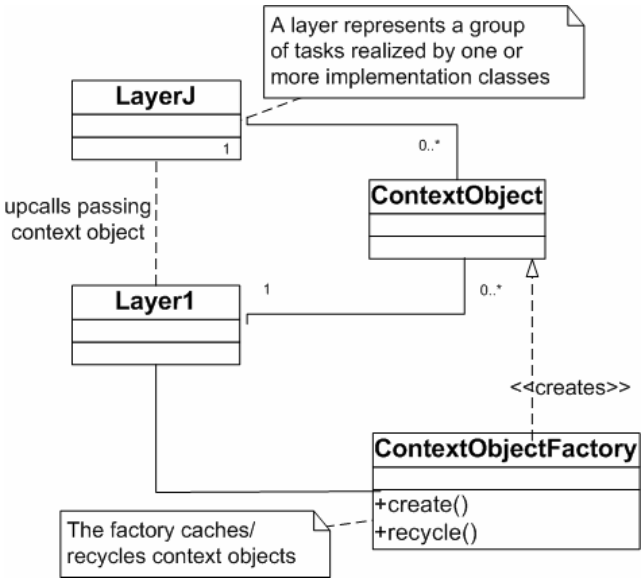


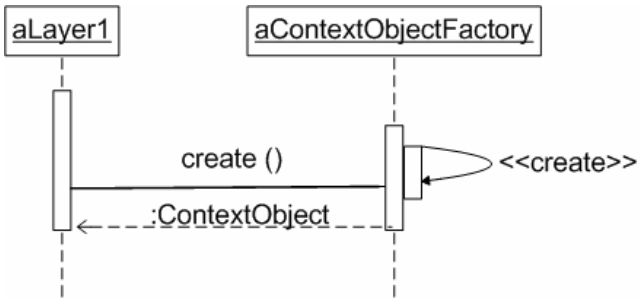
Figure 2. Structure of Context Object Pattern

of the peer and *operation ()* to set the name of the operation to invoke on the application provided servant.

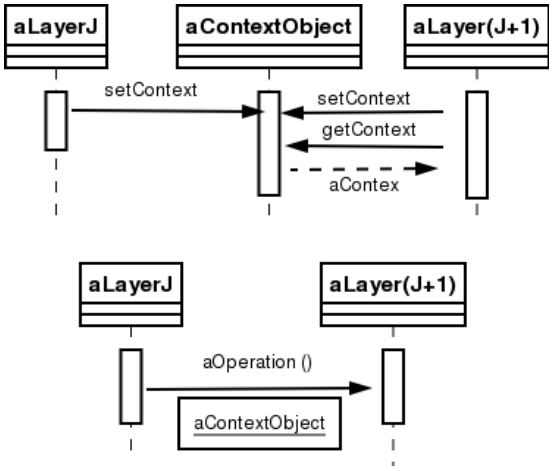
8 Dynamics

The following scenarios detail the interactions in the Context Object pattern.

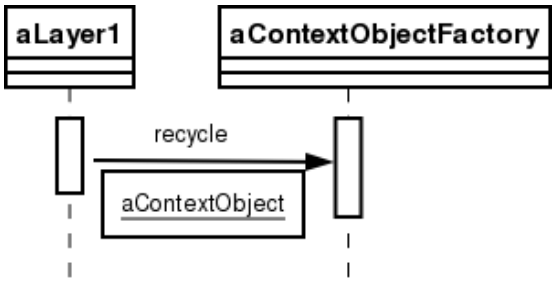
Scenario 1: Creation of a context object. A context object is created by the first layer, e.g., `Layer1` via the `create()` operation provided by the `ContextObjectFactory`. The following figure illustrates this scenario.



Scenario 2: Context addition and propagation. After a `ContextObject` has been created, `Layer1` adds context information required by `Layer2`. `Layer1` then upcalls Layer2 passing the `ContextObject`. `Layer2` (or any `LayerJ`, where J is not the last layer) follows a similar procedure of adding or obtaining context information stored in the object. It is important to note that the first layer, (`Layer1`) only adds context information, while the last layer (`Layern`) only retrieves context information associated with the ContextObject. The following figure illustrates this scenario.



Scenario 3: Reusing context objects. After one request/response processing cycle, `Layer1` uses the recycle operation provided on the `ContextObjectFactory`. This enables the factory to recycle the `ContextObject`. The following figure illustrates this scenario.



9 Implementation

The following steps are involved in implementing the Context Object pattern:

1. **Determine the information associated with a context object**. For each layer, determine the context information that is required by the next layer and provide operations to set/get that context information. Context information restricted to a single layer is not added to the Context Object. Use the Strategy pattern [GoF:95] to associate the right type of context information within the context object. The use of Strategy pattern also enables to accommodate variability. For example, middleware solutions support multiple protocols such as Internet Inter-ORB protocol (IIOP), User Datagram Protocol (UDP), and Stream Control Transfer Protocol (SCTP) [SCTP:01] for communication. Rather than create different types of context objects for each protocol, a middleware developer associates the context object with a base strategy class or interface that each protocol implements. At run-time this base strategy is associated with the concrete protocol used.

   → The TAO [TAO:98] CORBA ORB supports different protocols, such as Internet Inter-ORB protocol (IIOP), User Datagram Protocol (UDP), and Stream Control Transfer Protocol (SCTP) [SCTP:01] for communication. To transparently access each protocol, the Strategy pattern is used to associate the right protocol with the context object.

2. **Define the representation of context object.** The application developer should be unaware of the existence of context objects. Moreover, the context object representations should work seamlessly across different request types. The following are the two types of representations:

   - *Heap-allocated context objects*. In this approach, context objects are allocated dynamically using new operators in C++ and Java. Pointers in C++ and references in Java are then used to access (add or remove) context information within the objects.
   - *Stack-allocated context objects*. In this approach, context objects are created on the thread's stack rather than the heap. This approach is restricted to languages such as C++ that support stack allocation of non-primitive types.

   The heap-allocated approach eases the sharing of context objects across threads. In this approach, context objects can be put in queues shared by (1) producer/consumer threads and/or (2) asynchronous and synchronous processing layers as in a Half-Sync/Half-Async [POSA2:00] architectural pattern. The disadvantage of this approach is the need for dynamic allocation along the critical request processing path, which can be expensive. To alleviate this shortcoming, implementations can create a pool of context objects to service requests using an eager allocation strategy [POSA3:04]. A stack allocated approach is less expensive than heap allocation and is used in high performance C++ ORBs, such as TAO, to eliminate dynamic allocation along the critical path. An advantage of this approach is that it can be used in the Leader/Followers [POSA2:00] pattern that reduces dynamic (de)allocation, synchronization, and context switching when processing client requests. The disadvantage is the difficulty of sharing context objects across different threads.

3. **Determine how to pass Context Objects between the middleware layers.** This step is typically straightforward, *i.e.*, context objects can be passed either explicitly or implicitly. Explicit parameters are defined in signature of operations (as either pointers or references) used for communication between middleware layers. Implicit parameters are typically stored in a

thread specific storage [POSA2:00] or stored as environment accessed transparently by different layers. In the *Example Resolved* section, we show how to pass context objects explicitly.

10 Example Resolved

Consider a scenario where a CORBA server provides facilities for clients to query current stock prices. We refer to the server as a stock quoter. This server provides a `get_quote()` operation to get stock quotes. A client, using the server's object reference, invokes the `get_quote()` operation as follows:

```
long quote = obj_ref→get_quote ("Google");
```

Figure 4 illustrates how the demultiplexing process changes compared to Figure 1 when using a context object. A reactor listening on the server socket detects the request and reads the request onto an input buffer. The reactor notifies an event Handler (`IOEventHandler`) to further

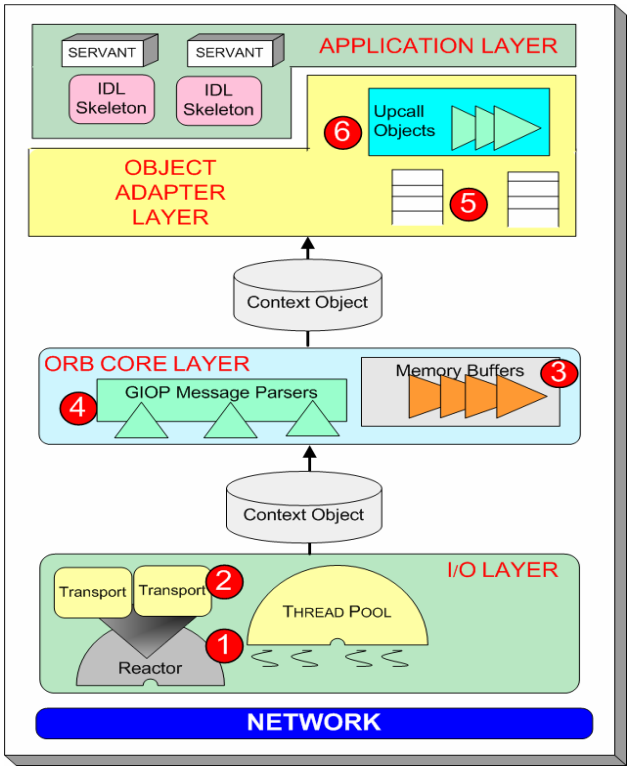


Figure 4: Layered CORBA demultiplexing using Context Object

process the request using the `process_request()` operation. This method creates a `Context_Object`, associating with it the transport (protocol specific (de) marshaling functionality and the buffer which stores the request message. Next, the event handler, uses the appropriate message parser to parse the message header.

This example illustrates how each layer adds/retrieves context information required/provided by the layer directly above/below it. Finally we describe the `Context_Object` class that encapsulates the information along with the appropriate accessor/mutator operations for individual data elements.

```
void
IOEventHandler::process_request ()
{
  // Use the Object Factory to create the
  // Context Object
  Context_Object server_request =
     this->factory_->
       create_context_object (transport_,
                              incoming_);

  // Parse the request header
  bool error =
     transport_->message_parser()->
         parse_header (server_request);

  // Raise exception if bad header
  if (error) throw (BAD_HEADER ());

  // Ask the next layer to handle request
  next_layer->dispatch (server_request);
}
```

This operation also populates the `Context_Object` with the CORBA service context information, end point information and the target operation name. This per request context information is stored in the context object for use by the next layers. If the request header information is parsed properly, the next layer (which in our case is the `ORB_Core` layer) can further process the request.

```
void
Message_Parser::parse_header
    (Context_Object &request)
{
  Input_Stream &input =
     request.incoming ();

  // Parse & populate the Context
  // information
  Service_Context & context =
    input.parse_context_information ();
  request.context (context);

  // Parse & populate end-point information
  // i.e. target host name port
  request.end_point_info
   (input.unmarshall_target_info ());

  // Parse and populate operation name
  request.operation
     (input.operation_name ());
}
```

The `ORB_Core` layer uses service context information stored in the Context_Object to take appropriate actions such as setting the right priority of the thread processing the request. This step shows how information stored in the

Context_Object by the lower layer is used in the next layer. This layer then parses (parse_object_key operation) the unique key information (ObjectKey) from the request and uses it to find the next (Object_Adapter) layer processing the request. The object key is associated with the context object for use in the next layer. The object adapter layer uses the context information populated by both the I/O and ORB layers to further process the request.

After the request has been processed completely, the call stack un-winds and the I/O layer sends the response back to the client.

```
void
ORB_Core::dispatch (Context_Object &request)
{
   // Process the context information
   this->
    process_context_info
     (request.service_context ());

  // Parse request to find Object key
  ObjectKey &key =
      parse_object_key (request);
  request.object_key (key);

  // Consult internal map to get the right
  // Object_Adapter to process the request
  Object_Adapter *adapter =
    this->map_[key];

  // Use this adapter to service request
  if (adapter)
   return adapter->dispatch (request);

  // Raise right exception
  throw (ADAPTER_NOT_FOUND ());
}


// Context Object class
class Context_Object {
public:

    /// Get/Set operation name
    const char *operation (void) const;
    void operation (const char *operation,
                    size_t length);

    /// Return the context information
    /// in the request/reply
    Service_Context []&
      service_context (void);

    /// Get and set the request id
    long request_id (void) const;
    void request_id (long req);

    /// Check if response is required
    bool is_response_expected () const;

    /// Get/Set Object Key information
    ObjectKey &object_key ();
    void object_key (const ObjectKey &key);

    /// Send reply to client
    void send_reply ();

 private:
    Transport *transport_;
    // Transport interface

    std::string operation_;
    // Operation name of operation

    Request_Context request_context_;
    // Service Context information

    ObjectKey object_key_ ;
    // Unique identifier for each request

    Profile end_point_;
    // Representation of target host port
    // details

};
```

11 Known Uses

Web servers and server pages engines provide context objects to efficiently share information for different requests within the same session. Due to the stateless design of HTTP, this is the only way to provide context information across otherwise independent HTTP requests. For example, login information and other security related information might be provided. Access to the context objects is enabled using techniques such as cookies (which are asynchronous completion tokens) or URL rewriting.

Scheduler activations are a mechanism where the kernel notifies an address spaces about kernel events affecting it [Andersen:92]. This approach allows an application to have complete knowledge of its scheduling state. Similarly, the user-level thread scheduler informs the kernel (via scheduler activations), of user level events the kernel should be aware of in making scheduling decisions. Scheduler activations are context objects used to exchange address space specific context information between the kernel and user level schedulers. Scheduler activations have also been implemented on the NetBSD [Nathan] operating system.

CORBA Request Processing. The TAO CORBA ORB [TAO] uses the Context Object pattern to demultiplex various types of requests and responses efficiently, scalably, and predictably. Upon receipt of a server request, a TAO server creates a CORBA-specified ServerRequest context object to hold the input and output buffers for the request. As the request is demultiplexed, successive layers add context information, such as the transport associated with the requests, the service context details, and protocol properties. Interceptors and fault-tolerance mechanisms use this context object to add information to the context object transparently. TAO also implements *service contexts*, which is another CORBA-specified mechanism for implicitly passing context information (such as priorities, security keys, and transaction identifiers) between a client and a server.

Theme Park tickets are human examples of the Context Object pattern. Consider a theme park that offers both free and paid rides. Visitors can choose packages that have varying prices based on the different rides chosen by visitors. The park uses a wrist-band as a ticket that has information about the ticket, price, and number of rides, etc. As a visitor proceeds through the park, her ticket is modified by a scanner that adds context to the band. Subsequently, all the information needed to validate her entry in the same/different ride is present in the band. The band acts as a context object that is efficient and easier for the park officials than having a database at each ride and tracking how many times the user has visited a particular ride.

12 Consequences

The Context Object pattern has the following **benefits:**

- It allows middleware developers to provide service for different request types without having to change the critical request processing path within the middleware.
- It enables efficient use of resources, such as request buffers, within the middleware since a context object can be recycled for different requests, thereby minimizing the number of dynamic allocations and reducing the amount of space needed.
- It eliminates the need for each middleware layer to store per request state. A context object can be associated with each thread processing the request, which alleviates the need to synchronize access to context information in each layer.
- It does not dictate any concurrency approach within the middleware, i.e., both synchronous and asynchronous request processing can be implemented at a server that uses context objects.

This pattern also incurs the following **liabilities:**

- Dynamically allocated context objects can degrade request processing latency due to (de)allocation for each request, which can also adversely affect latency and cause memory leaks if memory management is not done properly.
- As middleware implementations grow in functionality, the interface provided by a context object can become bloated. This can be ameliorated by using Decoupled Context Interface pattern [Hen:05]. This pattern allows the dependency between the context object and its type to be specified using interfaces thereby enabling different implementations of the Context Object to be created.
- The Context Object can grow in size if the context information added by a layer is not deleted by subsequent layers.

13 See Also

Existing patterns in the literature do not resolve all the aforementioned forces. For example, the *Chain of Responsibility* pattern [GOF:95] decouples the sender of a request/event from the actual component that handles the request/event by propagating the event along the hierarchy until a handler for the event is found. In this case, however, each layer performs some computation based on the context information and passes the request to the next layer in the hierarchy.

In the *Pipes and Filter* pattern [POSA1:96], data streams are transmitted via pipes to filters that transform the input generating modified output. Filters are relatively independent, i.e., an upstream filter makes as few assumptions as possible about the downstream filter and vice versa. In our case, however, a higher layer requires information from its lower layer. Moreover, correctness depends on the order in which the steps are executed (e.g., I/O → ORB core → object adapter).

In the *Asynchronous Completion Token* (ACT) [POSA2:00] pattern, a client invokes an asynchronous operation on a service. To help the client process the response from the service efficiently, it transmits an ACT which is a value that identifies the state necessary for the client to process the response, with the request. The response from the service also contains the same information, which is used by the client to find the completion handler efficiently. Similar to context objects, an asynchronous completion token (ACT) simplifies the data structure that a client needs to determine the right action. Context objects are also valid for only one request/response cycle, i.e., they are ephemeral, whereas multiple invocations of the same operation can have the same ACT.

In the *Encapsuate Context* [EncapsulateContext:03] pattern, a context container is used to hold all system or global data that would otherwise be scattered throughout different parts of the system. This pattern enables passing new system information into existing interfaces without having to add new parameters to a function's parameter list. Other patterns such as Role Specific/Partitioned Contexts [Hen:05] enable encapsulate context objects to be split according to usage roles. The Context object pattern is similar to the Encapsulate Context pattern since it enables new features to be added without having to modify existing interfaces. In addition, context objects provide a mechanism for accessing global/environment data and also enable addition/deletion of dynamic (per request) system information efficiently across different system layers.

In the *Variable Argument* [Zdun:05] pattern, the argument syntax allows for passing a variable number of parameters. In the *Optional Argument* pattern, special syntax is used to denote an argument as optional. A default value is supplied in case the client does not provide the argument. The *Non-positional* parameter passing pattern allows parameters to be passed in any order by enabling arguments to be passed as name/value pairs. The aforementioned patterns can be

used to implement a context object. For example, the variable argument pattern can be used to store different data on a per request basis. Similarly, the optional argument pattern can be used to provide a default value for context data in situations where a client does not provide one.

In the *Abstract Session* pattern [Abstract Session:97], a server maintains per client state in the form of a type safe session handle. The client uses this handle to access the service provided by the server. Although the Context Object pattern also maintains per-request client state at the server, unlike the Abstract Session pattern, a context object can be used to store context information across multiple layers. Similarly, the context information stored in the context object is typically valid only for one request/response cycle, whereas a handle used in the abstract session pattern is valid until the client closes or releases the session.

In the *Thread-Specific Storage* pattern [TSS:97], sequential operations access thread-specific state atomically without incurring locking overhead. The Context Object pattern can use thread-specific storage to enable different layers to access request-specific context information without incurring locking overhead.

14 Acknowledgments

We would like to thank Michael Kircher from Siemens Corporate Technology and Kevlin Henny for providing us valuable insights into other related patterns and how this pattern addresses forces not addressed by any one of the earlier patterns. We also would like to thank our PloP shepard Jorge L. Ortega Arjona and workshop reviewers for their constructive comments and suggestions that helped improve the paper.

References

[Abstract Session:97] Nat Pryce, Abstract Session Pattern, 4th Pattern Language of Program (PloP) Conference, 1997, Allterton Park, Illinois

[Anderson:92] Thomas E. Anderson and Brian N. Bershad and Edward D. Lazowska and Henry M. Levy, "Scheduler Activation: Effective Kernel Support for the User-Level Management of Parallelism," ACM Transactions on Computer Systems, Feb, 1992.

[CORBA:89] *The Common Object Request Broker: Architecture and Specification*, Object Management Group (OMG).

[CORBA-QoS:00] Douglas C. Schmidt and Steve Vinoski, "An Overview of the CORBA Messaging Quality of Service Framework," C++ Report, March 2000, Volume 12, Number 3.

[DCOM:98] *Distributed Component Object Model Protocol*, Microsoft Corporation, Jan 1998, Version 1.0, Microsoft Corporation

[DnC:03] Deployment and Configuration Adopted Submission, Object Management Group, OMG Document ptc/03-07-08, July 2003

[EncapsulateContext:03] Allen Kelly, Encapsulated Context Pattern, 10th European Conference on Pattern Languages of Program (EuroPloP), Irsee Germany, July 2003.

[GOF:95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[Gokhale:02] Aniruddha Gokhale and Balachandran Natarajan and Douglas C. Schmidt and Joseph K. Cross, "Towards Real-time Fault-Tolerant CORBA Middleware", Cluster Computing: the Journal on Networks, Software, and Applications, Special Issue on Dependable Distributed Systems, 2002.

[Hen:05] Kevlin Henney, "Context encapsulation - three stories, a language, and some sequences", In Proceedings of EuroPlop 2005, Irsee, Germany, July 2005.

[J2EE:01] *Java™ 2 Platform Enterprise Edition*, Sun Microsystems, 2001, java.sun.com/j2ee/index.html

[Williams:02] Nathan J Williams, "An Implementation of Scheduler Activations on the NetBSD Operating System," Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference (FREENIX '02), June 10-15, Monterey, CA.

[POSA1:96] Frank Buschmann and Regine Meunier and Hans Rohnert and Peter Sommerlad and Michael Stal, *Pattern-Oriented Software Architecture, Volume 1 – A System of Patterns*, Wiley & Sons, 1996..

[POSA2:00] Douglas Schmidt, Han Rohert, Michael Stal, and Frank Buschmann, *Pattern-Oriented Software Architecture, Volume 2 – Pattern for Concurrent and Networked Objects* Wiley & Sons, 2000.

[POSA3:04] M. Kircher and P. Jain: *Pattern-Oriented Software Architecture, Volume 3 – Patterns for Resouce Management*, John Wiley & Sons, 2004.

[RT-CORBA:00] Douglas C. Schmidt and Fred Kuhns, "An Overview of the Real-time CORBA Specification," IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing, June 200, Volume 33, Number 6.

[SCTP:01] Randall Stewart and Qiaobing Xie, *Stream Control Transmission Protocol (SCTP) A Reference Guide*, Addison-Wesley, 2001.

[Sommerlad:98] Peter Sommerlad, "The Manager Design Pattern", Pattern Languages of Program Design 3, Addison-Wesley, 1998.

[TAO:98] Douglas C. Schmidt, David Levine, and Sumedh Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications* Special Issue on Building Quality of Service into Distributed Systems, Elsevier Science, Volume 21, No. 4, April, 1998.

[TSS:97] Douglas C. Schmidt and Nat Pryce, "Thread Specific Storage C/C++", 4th Pattern Language of Program (PloP) Conference, 1997, Allterton Park, Illinois

[Zdun:05] Uwe Zdun, "Patterns of Argument Passing", In the Proceedings of Viking PloP Finland, September 2005.

Appendix A. Overview of Middleware Request Processing Steps

Consider a synchronous CORBA request:

```
result = obj_ref->operation (arg);
```

The sequence of steps required to process this request are described below[1] and illustrated in Figure 1.

1. On receipt of the request, the `Reactor` notifies a handler driven by a leader/followers thread pool. The other threads wait as followers on a condition variable or semaphore.
2. The leader thread reads the header of the request on connection to determine the size of the request. Each request is associated with a `Transport` that provides protocol specific connection handling.
3. A `Memory Buffer` is allocated from a memory pool to hold the request, after which the request data is read into the buffer.
4. A `Message Parser` examines the request to glean context information, e.g., the CORBA service context information (which can convey the request priority, security policies, and transaction ID), type of request, and total size of request.
5. The request is demarshaled to find the target servant and skeleton in the portable object adapter (POA); this process is aided by a unique object key created by the server and encoded in the request for every servant.
6. An upcall is dispatched to the application provided implementation and the reply if any is marshaled back. Finally, the reply is sent back to the client using the same incoming connection.

[1] This discussion has been generalized using the Reactor, Acceptor-Connector, and Leader/Follower [Leader-Follower] patterns from [POSA2].