Active Object

an Object Behavioral Pattern for
Concurrent Programming

R. Greg Lavender
G.Lavender@isode.com
ISODE Consortium Inc.
Austin, TX

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis

An earlier version of this paper appeared in a chapter in the book "Pattern Languages of Program Design 2" ISBN 0-201-89527-7, edited by John Vlissides, Jim Coplien, and Norm Kerth published by Addison-Wesley, 1996.

Abstract

This paper describes the Active Object pattern, which decouples method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. A broad class of producer/consumer and reader/writer applications are well-suited to this model of concurrency. This pattern is commonly used in distributed systems requiring multi-threaded servers. In addition, client applications, such as windowing systems and network browsers, employ active objects to simplify concurrent, asynchronous network operations.

1 Intent

The Active Object design pattern decouples method execution from method invocation to simplify synchronized access to an object that resides in its own thread of control. Methods invoked by clients on an object are not executed directly in the thread of the caller. Instead, they are transformed by a Proxy into Method Requests and stored in an Activation Queue. When synchronization constraints are met, Method Requests are executed by a Scheduler running in its own thread of control. Results can be returned to the client via a Future.

2 Also Known As

Concurrent Object and Actor

3 Example

To illustrate the Active Object pattern, consider the design of a communication Gateway [1]. A Gateway decouples cooperating components in a distributed system and allows them to interact without having direct dependencies among each other [2]. The Gateway shown in Figure 1 routes messages from one or more supplier processes to one or more consumer processes in a distributed system [3].

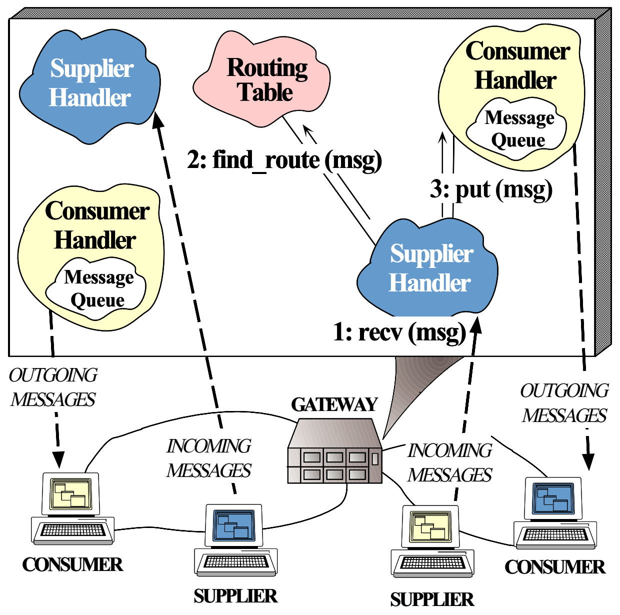


Figure 1: Communication Gateway

In our example, the Gateway, suppliers, and consumers communicate over TCP connections. Internally, the Gateway software contains a set of `Supplier` and `Consumer Handlers`, which act as local proxies [2, 4] for remote suppliers and consumers, respectively. `Supplier Handlers` receive messages from remote suppliers. They inspect address fields in the messages and use a routing table to find the appropriate `Consumer Handlers` associated with the one or more consumers. The `Consumer Handler`(s) then deliver the message to remote consumer(s).

The suppliers, consumers, and Gateway communicate using TCP, which is a connection-oriented protocol [5]. Therefore, `Consumer Handlers` in the Gateway software may encounter flow control from the TCP transport layer when they try to send data to a consumer. TCP uses flow control to ensure that fast suppliers or Gateways do not produce data more rapidly than slow consumers or congested networks can buffer and process the data.

To improve end-to-end quality of service (QoS) for all suppliers and consumers, a `Consumer Handler` must not block the entire Gateway waiting for flow control to abate over any one connection to a consumer. In addition, the Gateway must be able to scale up efficiently as the number of suppliers and consumers increase. An effective way to prevent blocking and to improve performance is to introduce concurrency into the design of the Gateway. Concurrent applications allow the thread of control of an object O that executes a method to be decoupled from the thread of control of objects that invoke methods on O.

4 Context

Applications that use concurrency to communicate between clients and servers running in separate threads of control.

5 Problem

Many applications benefit from applying concurrency to improve their QoS, *e.g.*, by enabling an application to handle multiple client requests in parallel. Instead of using single-threaded *passive objects*, which execute their methods in the thread of control of the client that invoked the method, concurrent objects reside in their own thread of control. However, if objects run concurrently we must synchronize access to their methods if these objects are shared by multiple clients. Three forces arise:

1. Methods invoked on the object should not block in order to prevent degrading the QoS of other methods: For instance, if a `Consumer Handler` object in our Gateway example is blocked due to flow control on its TCP connection, `Supplier Handler` objects should still be able to pass new messages to this `Consumer Handler`. Likewise, if other `Consumer Handlers` are *not* flow controlled, they should be able to send messages to their consumers independently of any blocked `Consumer Handlers`.

2. Synchronized access to shared data should be simple: Applications like the Gateway example are often hard to program if developers must explicitly use low-level synchronization mechanisms, such as acquiring and releasing locks. In general, methods that are subject to synchronization constraints should be serialized transparently when an object is accessed by multiple client threads.

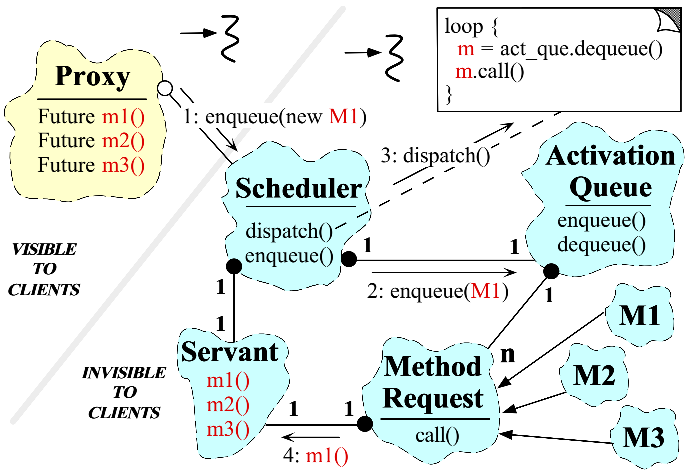
3. Applications should be designed to transparently leverage the parallelism available on a hardware/software platform: In our Gateway example, messages destined for different `Consumer Handlers` should be sent in parallel by a Gateway. If the entire Gateway runs in a single thread of control, however, performance bottlenecks cannot be alleviated transparently by running the Gateway on a multi-processor.

6 Solution

For each object that requires concurrent execution, decouple method invocation on the object from method execution. A *Proxy* [4, 2] represents the interface of the object and a *Servant* [6] provides its implementation. Both the Proxy and the Servant run in separate threads so that method invocation and method execution can run concurrency. At run-time, the Proxy transforms the client's method invocation into a *Method Request*, which is stored in an *Activation Queue* by a *Scheduler*. The Scheduler runs continuously in its own thread, dequeueing Method Request from the Activation Queue and dispatching them on the Servant that implements the object. Clients can obtain the results of a method's execution via the *Future* returned by the Proxy.

7 Structure

The structure of the Active Object pattern is illustrated in the following Booch class diagram:



There are six key participants in the Active Object pattern:

Proxy

- A Proxy [2, 4] provides an interface to publically accessible methods of an Active Object. This interface allows clients to invoke methods. When a client invokes a method defined by the Proxy, this triggers the construction and queueing of a Method Request.

Method Request

- A Method Request is constructed by a Proxy for any method call that requires synchronized access to an Active Object. Each Method Request maintains *context* *information*, such as method parameters and method code, necessary to (1) execute a method invocation and (2) to return any results of that invocation back through the Future. In addition, Method Requests may contain *predicates* that can be used to determine when the Method Request's synchronization constraints are met.

Activation Queue

- An Activation Queue maintains a priority queue of pending method invocations, which are represented as Method Request created by the Proxy.

Scheduler

- A Scheduler runs in its own thread managing an Activation Queue of Method Requests that are pending execution. The Scheduler decides which Method Request to dequeue and execute on the Servant that implements the method. This can be based on various criteria, such as *ordering*, *e.g.*, the order in which Method Requests are invoked by clients, and *synchronization constraints*, *e.g.*, mutual exclusion, which are evaluated by using Method Request predicates.

Servant

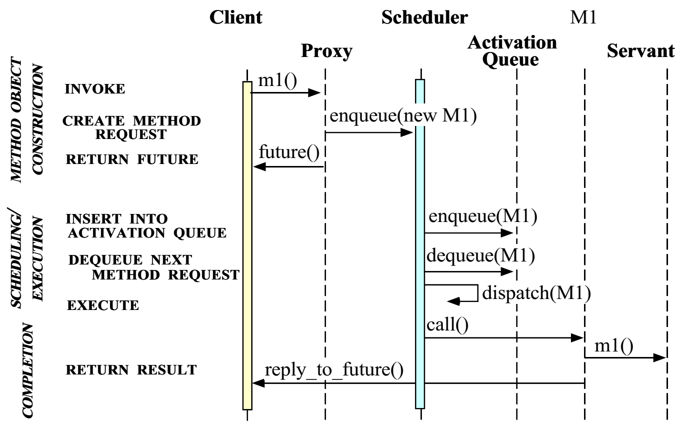
- A Servant defines the behavior and state that is being modeled as an Active Object. Servants implement the methods defined in the Proxy. In addition, Servants may provide methods used by Method Requests to implement predicates used by a Scheduler to determine Method Request execution order.

Future

- When a client invokes a method on the Proxy, a Future is returned to the client. The Future is an object that enforces "write-once, read-many" synchronization [7, 8]. It allows the caller to obtain the results of the method after the Servant completes the method execution.

8 Dynamics

The following figure illustrates the three phases of collaborations in the Active Object pattern:



1. Method Request construction and scheduling: In this phase, the client invokes a method defined by the Proxy. This triggers the creation of a Method Request, which maintains the argument bindings to the method, as well as any other bindings required to execute the method and return its results. The Proxy then passes the Method Request to the Scheduler, which enqueues it on the Activation Queue. If the method is defined as a *twoway* [6], a binding to a Future is returned to the caller of the method. If a method is defined as a *oneway*, *i.e.*, it has no return values, no Future is returned.

2. Method execution: In this phase, the Scheduler runs continuously in a different thread than any of its clients. Within this thread, the Scheduler monitors the Activation Queue and determines which Method Request(s) have become runnable, *e.g.*, when their synchronization constraints are met. When a Method Request becomes runnable, the Scheduler dequeues it, binds it to the Servant, and dispatches appropriate method on the Servant. When this method is called, it can access/update the state of its Servant and create its result(s).

3. Completion: In the final phase, the results, if any, are bound to the Future and the Scheduler continues to monitor the Activation Queue for runnable Method Requests. Clients use the Future to obtain the method's results after its execution completes. Any clients that rendezvous with the Future will obtain the results. The Method Request and Future can be garbage collected when they are no longer referenced.

9 Implementation

This section explains the steps involved in building a concurrent application using the Active Object pattern. The application implemented using the Active Object pattern is a portion of the Gateway from Section 3. Figure 2 illustrates the structure and participants in this example. The example in this section uses reusable components from the ACE framework [9]. ACE provides a rich set of reusable C++ wrappers and framework components that perform common communication software tasks across a wide range of OS platforms.

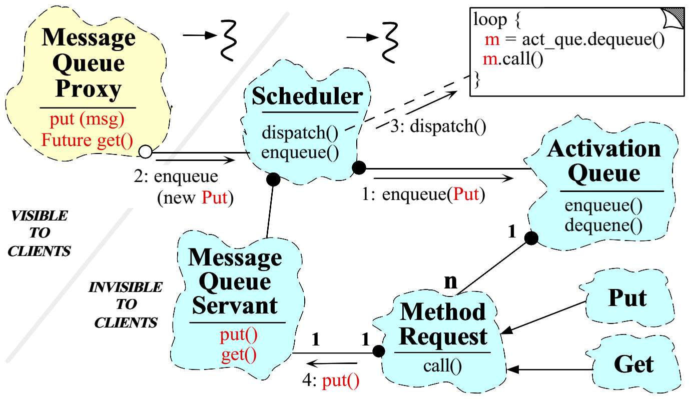


Figure 2: Implementing a Message Queue as an Active Object for `Consumer Handlers`

The following steps illustrate how to implement a `Message Queue` as an Active Object in the Gateway. `Consumer Handlers` use `Message Queues` to obtain messages from `Supplier Handlers` in the Gateway and send them to their remote consumers.

1. Implement the Servant: A Servant defines the behavior and state that is being modeled as an Active Object. The mehtods a Servant implements are accessible by clients via a Proxy. In addition, a Servant may contain other methods that Method Requests can use to implement predicates that allow a Scheduler to evaluate run-time synchronization constraints. These constraints determine the order in which a Scheduler dispatches Method Requests.

In our Gateway example, the Servant is a message queue that allows `Consumer Handlers` to obtain messages from `Supplier Handlers` and to send them to their corresponding remote consumers. The following class provides an interface for this Servant:

```
class Message_Queue_Servant
{
public:
  Message_Queue_Servant (size_t size);

  // Predicates.
  bool empty (void) const;
  bool full (void) const;

  // Queue operations.
  void put (const Message x);
  Message get (void);

private:
  // Internal Queue representation.
};
```

The `empty` and `full` predicates distinguish three internal states: (1) empty, (2) full, and (3) neither empty nor full. The `put` and `get` methods implement the insertion and removal operations on the queue, respectively.

Note that the Active Object pattern is designed so that synchronization mechanisms can remain external to the Servant. Therefore, methods in the `Message_Queue_Servant` class do not include any code that implements synchronization or mutual exclusion. It only provides methods that implement the Servant's functionality. This design avoids the *inheritance anomaly* [10, 11, 12, 13] problem, which inhibits the reuse of Servant implementations by subclasses that require specialized or different synchronization policies.

2. Implement the Proxy and Method Requests: The Proxy provides clients with an interface to the Servant's methods. For each method invocation by a client, the Proxy creates a Method Request. A Method Request is an abstraction for the context[1] of a method. This context typically includes the method parameters, a binding to the Servant the method will be applied to, a Future for the result, and the code for the Method Request's `call` method.

In our Gateway example, the `Message_Queue_Proxy` provides an abstract interface to the `Message_Queue_Servant` defined in Step 1. This message queue is used by a `Consumer Handler` to queue messages for delivery to consumers, as shown in Figure 1. In addition, the `Message_Queue_Proxy` is a factory that constructs instances of Method Requests and passes them to a Scheduler, which queues them for subsequent execution. The C++ implementation for the `Message_Queue_Proxy` is shown below:

```
class Message_Queue_Proxy
{
public:
  // Bound the message queue size.
  enum { MAX_SIZE = 100 };

  Message_Queue_Proxy
    (size_t size = MAX_SIZE) {
    scheduler_ = new MQ_Scheduler;
    servant_ = new Message_Queue_Servant (size);
  }

  // Schedule <put> to run as an active object.
  void put (const Message m) {
    Method_Request *method =
      new Put (servant_, m);
    scheduler_->enqueue (method);
  }

  // Return a Message_Future as the ''future''
  // result of an asynchronous <get>
  // method on the active object.
  Message_Future get (void) {
    Message_Future result;

    Method_Request *method =
      new Get (servant_, result);
    scheduler_->enqueue (method);
    return result;
  }

  // These predicates can execute directly
  // since they are "const".
  bool empty (void) const {
    return servant_->empty ();
  }
  bool full (void) const {
    return servant_->full ();
  }
```

[1] This context is often called a *closure*.

```
protected:
  // The Servant that implements the
  // Active Object methods.
  Message_Queue_Servant *servant_;

  // A scheduler for the Message Queue.
  MQ_Scheduler *scheduler_;
};
```

Each method of a `Message_Queue_Proxy` transforms its invocation into a Method Request and passes the request to the `MQ_Scheduler`, which enqueues it for subsequent activation. A `Method_Request` base class defines virtual `guard` and `call` methods that are used by a Scheduler to determine if a Method Request can be executed and to execute the Method Request on its Servant, respectively, as follows:

```
class Method_Request
{
public:
  // Evaluate the synchronization constraint.
  virtual bool guard (void) const = 0;

  // Implement the method.
  virtual void call (void) = 0;
};
```

The methods in this class must be defined by subclasses, one for each method defined in the Proxy. For instance, when a client invokes the `put` method on the Proxy in our Gatway example, this method is transformed into an instance of the `Put` subclass, which inherits from `Method_Request` and contains a pointer to the `Message_Queue_Servant`, as follows:

```
class Put : public Method_Request
{
public:
  Put (Message_Queue_Servant *rep,
       Message arg)
    : servant_ (rep), arg_ (arg) {}

  virtual bool guard (void) const {
    // Synchronization constraint.
    return !servant_->full ();
  }

  virtual void call (void) {
    // Enqueue message into message queue.
    servant_->put (arg_);
  }

private:
  Message_Queue_Servant *servant_;
  Message arg_;
};
```

A run-time binding to the `Message_Queue_Servant` is used by the `call` method, which is similar to providing a "`this`" pointer to a C++ method. The method is executed in the context of that Servant representation, as shown in the `Put::call` method above.

The Proxy also transforms the `get` method into an instance of the `Get` class, which is defined as follows:

```
class Get : public Method_Request
{
public:
  Get (Message_Queue_Servant *rep,
       Message_Future &f)
    : servant_ (rep), result_ (f) {}

  bool guard (void) const {
    // Synchronization constraint.
    return !servant_->empty ();
  }

  virtual void call (void) {
    // Bind the dequeued message to the
    // future result object.
    result_ = servant_->get ();
  }

private:
  Message_Queue_Servant *servant_;

  // Message_Future result value.
  Message_Future result_;
};
```

For every method in the Proxy that returns a value, such as the `get` method in our Gateway example, a `Message_Future` is returned to the client thread that calls it, as shown in implementation Step 4 below. The caller may choose to evaluate the `Message_Future`'s value immediately. Conversely, the evaluation of a return result from a method invocation on an Active Object can be deferred.

3. Implement the Activation Queue and Scheduler: Each Method Request is enqueued on an Activation Queue, which is typically implemented as a priority queue, as follows:

```
class Activation_Queue
{
public:
  typedef ... iterator;

  // Insert <method> into the queue.
  void enqueue (Method_Request *method);

  // Remove <method> into the queue.
  void dequeue (Method_Request *method);

private:
  // ...
};
```

In addition, an `Activation_Queue` offers an iterator for traversing its elements, which is typically implemented using the Iterator pattern [4].

The `Activation_Queue` is used by a Scheduler to enforce specific synchronization constraints on behalf of a Servant. To accomplish this, the Scheduler uses `Method_Request` guards to determine which `Method_Request`s to execute. In turn, these guards use predicates defined in Servant implementations to represent different states the Servant can be in.

For instance, in our Gateway example, the `MQ_Scheduler` determines the order to process `put` and `get` methods based on predicates of the underlying `Message_Queue_Servant`. These predicates reflect the state of the Servant, such as whether the message queue is empty, full, or neither.

The `MQ_Scheduler` accesses the `Message_Queue_Servant` `empty` and `full` predicates via the `Method_Request` `guard`s when it evaluates synchronization constraints prior to executing a `put` or `get` method on the `Message_Queue_Servant`. The use of constraints ensures fair shared access to the `Message_Queue_Servant`.

The `MQ_Scheduler` enqueues `Put` and `Get` instances into an `Activation_Queue` as follows:

```
class MQ_Scheduler
{
public:
  // ... constructors/destructors, etc.,

  // Insert the Method Request into
  // the Activation_Queue.  This method
  // runs in the thread of its caller.
  void enqueue (Method_Request *method) {
    act_que_->enqueue (method);
  }

  // Dispatch the Method Requests
  // on their Servant.  This method
  // runs in a separate thread.
  virtual void dispatch (void);

protected:
  // Queue of pending Method_Requests.
  Activation_Queue *act_que_;
};
```

In general, a Scheduler executes its `dispatch` method in a thread of control that is different from its clients' threads. Within this thread, the Scheduler monitors its `Activation_Queue`. The Scheduler selects a `Method_Request` whose *guard* evaluates to "true," *i.e.*, whose synchronization constraints are met. This `Method_Request` is then executed by invoking its `call` hook method, as follows:

```
virtual void
MQ_Scheduler::dispatch (void)
{
  // Iterate continuously in a
  // separate thread.
  for (;;) {
    Activation_Queue::iterator i;

    for (i = act_que_->begin ();
         i != act_que_->end ();
         i++) {
      // Select a Method Request 'm'
      // whose guard evaluates to true.
      Method_Request *m = *i;

      if (m->guard ()) {
        // Remove <m> from the queue first
        // in case <call> throws an exception.
        act_que_->dequeue (m);
        m->call ();
      }
    }
  }
}
```

The `MQ_Scheduler`'s `dispatch` implementation executes the first `Method_Request` whose `guard` predicate evaluates to true. However, a Scheduler implementation could also check which of all pending invocations could be executed and then select a subset of these according to criteria such as mutual exclusion or invocation order.

In general, a Scheduler may contain variables that represent the synchronization state of the Servant. The variables defined depend on the type of synchronization mechanism that is required. For example, with reader-writer synchronization, counter variables may be used to keep track of the number of read and write requests. In this case, the values of the counters are independent of the state of the Servant since they are only used by the scheduler to enforce the correct synchronization policy on behalf of the Servant.

4. Determine rendezvous and return value policies: The rendezvous policy determines how clients obtain return values from methods invoked on active objects. A rendezvous policy is required since active objects do not execute in the same thread as clients that invoke their methods. Different implementations of the Active Object pattern typically choose from the following rendezvous and return value policies:

- *Synchronous waiting* – Block the caller synchronously at the Proxy until the active object accepts the method call.
- *Synchronous timed wait* – Block only for a bounded amount of time and fail if the active object does not accept the method call within that period. If the timeout is zero this scheme is referred to as "polling."
- *Asynchronous* – Queue the method call and return control to the caller immediately. If the method produces a result value then some form of Future mechanism must be used to provide synchronized access to the value (or the error status if the method fails).

A Future allows asynchronous invocations that return a value to the caller. When a Servant completes the method execution, it acquires a write lock on the Future and updates the Future with a result value of the same type as that used to parameterize the Future. Any readers of the result value that are currently blocked waiting for the result value are awakened and may access the result value concurrently. A Future object can be garbage collected after the writer and all readers no longer reference the Future.

In our Gateway example, the `get` method in the `Message_Queue_Proxy` is ultimately processed by the `Get::call` method, which binds the `get` operation result to the `result_` Future as shown in Step 2 above. Since the `Message_Queue_Proxy` `get` method returns a value, a `Message_Future` is returned when the client calls it. The `Message_Future` is defined as follows:

```
class Message_Future
public:
  // ... constructors/destructors, etc.,

  // Type conversion, which blocks
  // waiting to obtain the result of the
  // asynchronous method invocation.
  operator Message ();
};
```

The client can obtain the Message result value from a Message_Future object in either of the followings ways:

• **Immediate evaluation:** The caller may choose to evaluate the Message_Future's value immediately. For example, a Gateway Consumer Handler running in a separate thread may choose to block until new messages arrive from Supplier Handlers, as follows:

```
  Message_Queue_Proxy mq;
  // ...

  // Conversion of Message_Future from the
  // get() method into a Message causes the
  // thread to block until a message is
  // available on the mq.
  Message msg = mq.get ();

  // Transmit message to the consumer.
  send (msg);
```

• **Deferred evaluation:** The evaluation of a return result from a method invocation on an Active Object can be deferred. For example, if messages are not available immediately, a Consumer Handler can store the Message_Future return value from mq and perform other "bookkeeping" tasks, such as exchanging *keepalive messages* to ensure its consumer is still active. When the Consumer Handler is done with these tasks it can block until a message arrives from an Supplier Handler, as follows:

```
  // Obtain a future (does not block the caller).
  Message_Future future = mq.get ();

  // Do something else here...

  // Evaluate future in the conversion operator;
  // may block if the result is not available yet.
  Message msg = Message (future);
```

10 Variants

The following are variations of the Active Object pattern.

Workpile: The Workpile is a generalization of Active Object that supports multiple Servants per Active Object. These Servants can offer the same services to increase QoS. Every Servant runs in its own thread and actively ask the Scheduler to assign a new request when it is ready with its current job. The Scheduler then assigns a new job as soon as one is available.

Integrated Scheduler: For simplicity, the roles of the Proxy and Servant are often integrated into the Scheduler component. Moreover, the transformation of the method call into a Method Request can also be integrated into the Scheduler. For instance, the following is another way to implement the Message Queue example:

```
class MQ_Scheduler
public:
  // ... constructors/destructors, etc.,

  void put (const Message m) {
    Method_Request *method =
      new Put (servant_, m);
    queue_->enqueue (method);
  }

  Message_Future get (void) {
    Message_Future result;

    Method_Request *method =
      new Get (servant_, result);
    queue_->enqueue (method);
    return result;
  }

  // ...
protected:
  Message_Queue_Servant *servant_;

  // ...
};
```

By centralizing where Method Requests are generated, the pattern implementation can be simplified since the Servant needn't be coupled with the Proxy.

Polymorphic Futures: A Polymorphic Future [14] allows parameterization of the eventual result type represented by the Future and enforces the necessary synchronization. A typed future result value provides write-once, read-many synchronization. Whether a caller blocks on a future depends on whether or not a result value has been computed. Hence, a Future is partly a reader-writer condition synchronization pattern and partly a producer-consumer synchronization pattern.

11 Known Uses

The following are specific known uses of this pattern:

CORBA ORBs: The Active Object pattern has been used to implement concurrent ORB middleware frameworks, such as CORBA [6] and DCOM [15]. For instance, the TAO ORB [16] implements the Active Object pattern for its default concurrency model [17]. In this design, CORBA stubs correspond to the Active Object pattern's Proxies, which transform remote operation invocations into CORBA Requests. The TAO ORB Core's Reactor is the Scheduler and the socket queues in the ORB Core correspond to the Activation Queues. Developers create Servants that execute the methods in the context of the server. Clients can either make synchronous twoway invocations, which block the calling thread until the operation returns, or they can make asynchronous method invocations, which return a Future that can be evaluated at a later point.

ACE Framework: Reusable implementations of the Method Request, Activation Queue, and Future components in the Active Object pattern are provided in the ACE framework [9]. These components have been used to implement many production distributed systems.

Siemens MedCom: The Active Object pattern is used in the Siemens MedCom framework, which provides a black-box component-oriented framework for electronic medical systems [18]. MedCom employ the Active Object pattern to simplify client windowing applications that access patient information on various medical servers.

Siemens Call Center management system: This system uses the Workpile variant of the Active Object pattern.

Actors: The Active Object pattern has been used to implement Actors [19]. An Actor contains a set of instance variables and behaviors that react to messages sent to an Actor by other Actors. Messages sent to an Actor are queued in the Actor's message queue. In the Actor model, messages are executed in order of arrival by the "current" behavior. Each behavior nominates a replacement behavior to execute the next message, possibly before the nominating behavior has completed execution. Variations on the basic Actor model allow messages in the message queue to be executed based on criteria other than arrival order [20]. When the Active Object pattern is used to implement Actors, the Scheduler corresponds to the Actor scheduling mechanism, Method Request correspond to the behaviors defined for an Actor, and the Servant is the set of instance variables that collectively represent the state of an Actor [21]. The Proxy is simply a strongly-typed mechanism used to pass a message to an Actor.

12 Consequences

The Active Object pattern provides the following benefits:

Enhance application concurrency and simplify synchronization complexity: Concurrency is enhanced by allowing method invocations on active objects to execute simultaneously. Synchronization complexity is simplified by the Active Object Scheduler, which evaluates synchronization constraints to guarantee serialized access to Servants, depending on the state of the resource.

Transparently leverage available parallelism: If the hardware and software platforms support multiple CPUs efficiently, this pattern can allow multiple active objects to execute in parallel, subject to their synchronization constraints.

Method execution order can differ from method invocation order: Methods invoked asynchronously are executed based on their synchronization constraints, which may differ from their order of invocation.

However, the Active Object pattern has the following liabilities:

Performance overhead: Depending on how the Scheduler is implemented, *e.g.*, in user-space vs. kernel-space, context switching, synchronization, and data movement overhead may occur when scheduling and executing active object method invocations. In general, the Active Object pattern is most applicable on relatively coarse-grained objects. In contrast, if the objects are very fine-grained, the performance overhead of active objects can be high [22].

Complicated debugging: It may be difficult to debug programs containing active objects due to the concurrency and non-determinism of the Scheduler. Moreover, many debuggers do not support concurrent applications adequately.

13 See Also

The Mutual Exclusion (Mutex) pattern [23] is a simple locking pattern that can occur in slightly different forms, such as a spin lock or a semaphore. The Mutex pattern can have subtle semantics, such as recursive mutexes and priority mutexes.

The Consumer-Producer Condition Synchronization pattern is a common pattern that occurs when the synchronization policy and the resource are related by the fact that synchronization is dependent on the state of the resource.

The Reader-Writer Condition Synchronization pattern is a common synchronization pattern that occurs when the synchronization mechanism is not dependent on the state of the resource. A readers-writers synchronization mechanism can be implemented independent of the type of resource requiring reader-writer synchronization.

The Reactor pattern [24] is responsible for demultiplexing and dispatching of multiple event handlers that are triggered when it is possible to initiate an operation without blocking. This pattern is often used in lieu of the Active Object pattern in order to schedule callback operations to passive objects. It can also be used in conjunction of the Reactor pattern to form the Half-Sync/Half-Async pattern described in the next paragraph.

The Half-Sync/Half-Async pattern [25] is an architectural pattern that decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency. This pattern typically uses the Active Object pattern to implement the Synchronous task layer, the Reactor pattern [24] to implement the Asynchronous task layer, and a Producer/Consumer pattern to implement the Queueing layer.

Acknowledgements

The genesis for the Active Object pattern originated with Greg Lavender. Thanks to Frank Buschmann for extensive comments that greatly improved the form and content of this version of the pattern description.

References

[1] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

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

[3] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

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

[5] W. R. Stevens, *TCP/IP Illustrated, Volume 1*. Reading, Massachusetts: Addison Wesley, 1993.

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

[7] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 501–538, Oct. 1985.

[8] B. Liskov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 260–267, June 1988.

[9] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[10] P. America, "Inheritance and Subtyping in a Parallel Object-Oriented Language," in *ECOOP'87 Conference Proceedings*, pp. 234–242, Springer-Verlag, 1987.

[11] D. G. Kafura and K. H. Lee, "Inheritance in Actor-Based Concurrent Object-Oriented Languages," in *ECOOP'89 Conference Proceedings*, pp. 131–145, Cambridge University Press, 1989.

[12] S. Matsuoka, K. Wakita, and A. Yonezawa, "Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages," *OOPS Messenger*, 1991.

[13] M. Papathomas, "Concurrency Issues in Object-Oriented Languages," in *Object Oriented Development* (D. Tsichritzis, ed.), pp. 207–245, Centre Universitaire D'Informatique, University of Geneva, 1989.

[14] R. G. Lavender and D. G. Kafura, "A Polymorphic Future and First-Class Function Type for Concurrent Object-Oriented Programming in C++," in *Forthcoming*, 1995. http://www.cs.utexas.edu/users/lavender/papers/futures.ps.

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

[16] 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.

[17] D. C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[18] P. Jain, S. Widoff, and D. C. Schmidt, "The Design and Performance of MedJava – Experience Developing Performance-Sensitive Distributed Applications with Java," *IEE/BCS Distributed Systems Engineering Journal*, 1998.

[19] G. Agha, *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[20] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled-Sets," in *OOPSLA'89 Conference Proceedings*, pp. 103–112, Oct. 1989.

[21] D. Kafura, M. Mukherji, and G. Lavender, "ACT++: A Class Library for Concurrent Programming in C++ using Actors," *Journal of Object-Oriented Programming*, pp. 47–56, October 1992.

[22] D. C. Schmidt and T. Suda, "Measuring the Performance of Parallel Message-based Process Architectures," in *Proceedings of the Conference on Computer Communications (INFOCOM)*, (Boston, MA), pp. 624–633, IEEE, April 1995.

[23] Paul E. McKinney, "A Pattern Language for Parallelizing Existing Programs on Shared Memory Multiprocessors," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[24] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[25] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O," in *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, September 1995.