Monitor Object

An Object Behavioral Pattern for
Concurrent Programming

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

1 Intent

The Monitor Object pattern synchronizes method execution to ensure only one method runs within an object at a time. It also allows an object's methods to cooperatively schedule their execution sequences.

2 Also Known As

Thread-safe Passive Object

3 Example

Consider the design of the communication Gateway shown in Figure 1. The Gateway process contains multiple supplier

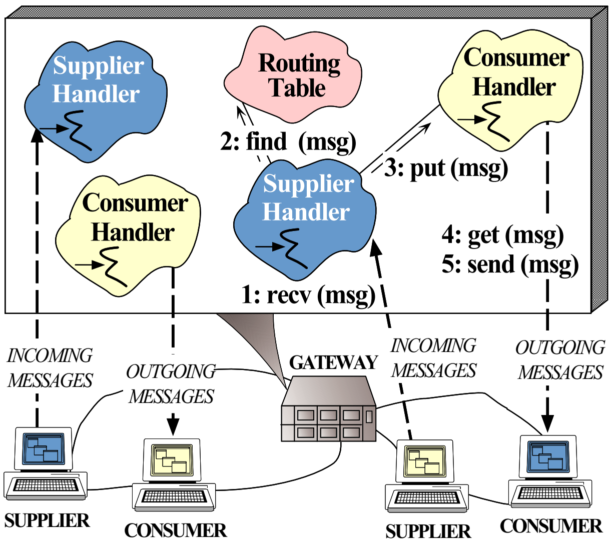


Figure 1: Communication Gateway

handler and consumer handler objects that run in separate threads and route messages from one or more remote suppliers to one or more remote consumers, respectively. When a supplier handler thread receives a message from a remote supplier, it uses an address field in the message to determine the corresponding consumer handler, whose thread then delivers the message to its remote consumer.

When suppliers and consumers reside on separate hosts, the Gateway uses the connection-oriented TCP [1] protocol to provide reliable message delivery and end-to-end flow control. TCP's flow control algorithm blocks fast senders when they produce messages more rapidly than slower receivers can process the messages. The entire Gateway should not block, however, while waiting for flow control to abate on outgoing TCP connections. To minimize blocking, therefore, each consumer handler can contain a thread-safe message queue that buffers new routing messages it receives from its supplier handler threads.

One way to implement a thread-safe `Message Queue` is to use the Active Object pattern [2], which decouples the thread used to invoke a method from the thread used to execute the method. As shown in Figure 2, each message queue active object contains a bounded buffer and its own thread of control that maintains a queue of pending messages. Using

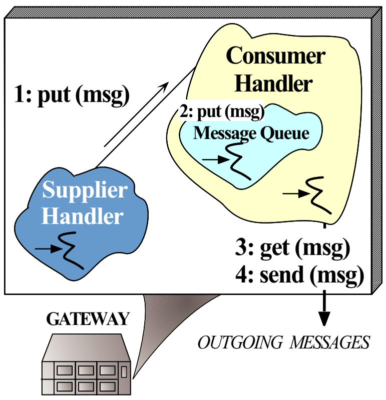


Figure 2: Implementing Message Queues as Active Objects

the Active Object pattern to implement a thread-safe message queue decouples supplier handler threads in the Gateway process from consumer handler threads so all threads can run concurrently and block independently when flow control occurs on various TCP connections.

Although the Active Object pattern can be used to implement a functional Gateway, it has the following drawbacks:

Performance overhead: The Active Object pattern provides a powerful concurrency model. It not only synchronizes concurrent method requests on an object, but also can perform sophisticated scheduling decisions to determine the order in which requests execute. These features incur non-trivial amounts of context switching, synchronization, dynamic memory management, and data movement overhead, however, when scheduling and executing method requests.

Programming overhead: The Active Object pattern requires programmers to implement up to six components: *proxies*, *method requests*, an *activation queue*, a *scheduler*, a *servant*, and *futures* for each proxy method. Although some components, such as activation queues and method requests, can be reused, programmers may have to reimplement or significantly customize these components each time they apply the pattern.

In general, the performance and programming overhead outlined above can be unnecessarily expensive if an application does not require all the Active Object pattern features, particularly its sophisticated scheduling support. Yet, programmers of concurrent applications must ensure that certain method requests on objects are synchronized and/or scheduled appropriately.

4 Context

Applications where multiple threads of control access objects simultaneously.

5 Problem

Many applications contain objects that are accessed concurrently by multiple client threads. For concurrent applications to execute correctly, therefore, it is often necessary to synchronize and schedule access to these objects. In the presence of this problem, the following three requirements must be satisfied:

1. Synchronization boundaries should correspond to object methods: Object-oriented programmers are accustomed to accessing an object only through its interface methods in order to protect an object's data from uncontrolled changes. It is relatively straightforward to extend this object-oriented programming model to protect an object's data from uncontrolled concurrent changes, known as *race conditions*. Therefore, an object's method interface should define its synchronization boundaries.

2. Objects, not clients, should be responsible for their own method synchronization: Concurrent applications are harder to program if clients must explicitly acquire and release low-level synchronization mechanisms, such as semaphores, mutexes, or condition variables. Thus, objects should be responsible for ensuring that any of their methods requiring synchronization are serialized transparently, *i.e.*, without explicit client intervention.

3. Objects should be able to schedule their methods cooperatively: If an object's methods must block during their execution, they should be able to voluntarily relinquish their thread of control so that methods called from other client threads can access the object. This property helps prevent deadlock and makes it possible to leverage the concurrency available on hardware/software platforms.

6 Solution

For each object accessed concurrently by client threads define it as a *monitor object*. Clients can access the services defined by a monitor object only through its *synchronized methods*. To prevent race conditions involving monitor object state, only one synchronized method at a time can run within a monitor object. Each monitored object contains a *monitor lock* that synchronized methods use to serialize their access to an object's behavior and state. In addition, synchronized methods can determine the circumstances under which they suspend and resume their execution based on one or more *monitor conditions* associated with a monitor object.

7 Structure

There are four participants in the Monitor Object pattern:

Monitor object

- A monitor object exports one or more methods to clients. To protect the internal state of the monitor object from uncontrolled changes or race conditions, all clients must access the monitor object only through these methods. Each method executes in the thread of the client that invokes it because a monitor object does not have its own thread of control.[1]

  For instance, the consumer handler's message queue in the Gateway application can be implemented as a monitor object.

Synchronized methods

- Synchronized methods implement the thread-safe services exported by a monitor object. To prevent race conditions, only one synchronized method can execute within a monitor at any point in time, regardless of the number of threads that invoke the object's synchronized methods concurrently or the number of synchronized methods in the object's class.

  For instance, the `put` and `get` operations on the consumer handler's message queue should be synchronized methods to ensure that routing messages can be inserted and removed simultaneously by multiple threads without corrupting a queue's internal state.

[1] In contrast, an active object [2] *does* have its own thread of control.

Monitor lock

- Each monitor object contains its own monitor lock. Synchronized methods use this monitor lock to serialize method invocations on a per-object basis. Each synchronized method must acquire/release the monitor lock when the method enters/exits the object, respectively. This protocol ensures the monitor lock is held whenever a method performs operations that access or modify its object's state.

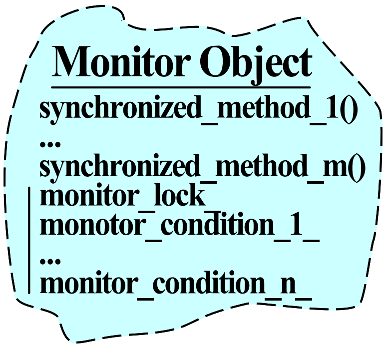
  For instance, a `Thread_Mutex` [3] could be used to implement the message queue's monitor lock.

Monitor condition

- Multiple synchronized methods running in separate threads can cooperatively schedule their execution sequences by waiting for and notifying each other via monitor conditions associated with their monitor object. Synchronized methods use monitor conditions to determine the circumstances under which they should suspend or resume their processing.

  For instance, when a consumer handler thread attempts to dequeue a routing message from an empty message queue, the queue's `get` method must release the monitor lock and suspend itself until queue is no longer empty, *i.e.*, when a supplier handler thread inserts a message in it. Likewise, when a supplier handler thread attempts to enqueue a message into a full queue, the queue's `put` method must release the monitor lock and spend itself until the queue is no longer full, *i.e.*, when a consumer handler removes a message from it. A pair of POSIX condition variables [4] can be used to implement the message queue's *not-empty* and *not-full* monitor conditions.

The structure of the Monitor Object pattern is illustrated in the following UML class diagram:



8 Dynamics

The following collaborations occurs between participants in the Monitor Object pattern.

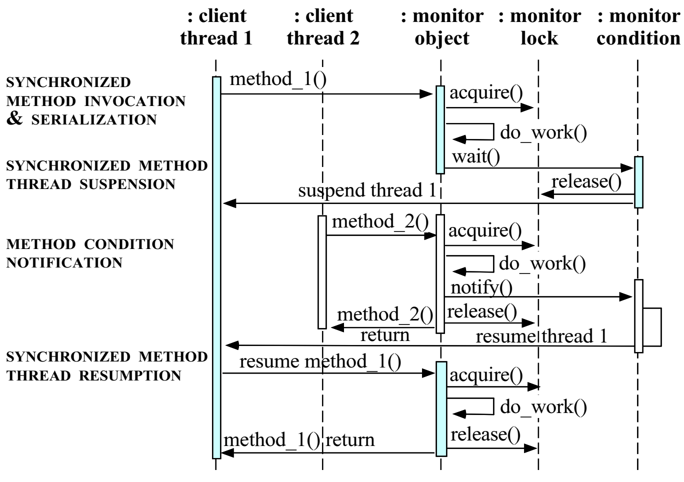
1. Synchronized method invocation and serialization: When a client invokes a synchronized method on a monitor object, the method must first acquire its monitor lock. A monitor lock cannot be acquired as long as another synchronized method is executing within the monitor object. In this case, the client thread will block until it acquires the monitor lock, at which point the synchronized method will acquire the lock, enter its critical section, and perform the service implemented by the method. Once the synchronized method has finished executing, the monitor lock must be released so that other synchronized methods can access the monitor object.

2. Synchronized method thread suspension: If a synchronized method must block or cannot otherwise make immediate progress, it can *wait* on one of its monitor conditions, which causes it to "leave" the monitor object temporarily [5]. When a synchronized method leaves the monitor object, the monitor lock is released automatically and the client's thread of control is suspended on the monitor condition.

3. Method condition notification: A synchronized method can *notify* a monitor condition in order to resume a synchronized method's thread that had previously suspended itself on the monitor condition. In addition, a synchronized method can notify *all* other synchronized methods that previously suspended their threads on a monitor condition.

4. Synchronized method thread resumption: Once a previously suspended synchronized method thread is notified, its execution can resume at the point where it waited on the monitor condition. The monitor lock is automatically reacquired before the notified thread "reenters" the monitor object and resumes executing the synchronized method.

The following figure illustrates the collaborations in the Monitor Object pattern:



9 Implementation

The following steps illustrate how to implement the Monitor Object pattern.

1. Define the monitor object's interface methods: The interface of a monitor object exports a set of methods to clients. Interface methods are typically *synchronized*, *i.e.*, only one of them at a time can execute in a particular monitor object.

In our Gateway example, each consumer handler contains a message queue and a TCP connection. The message queue can be defined as a monitor object that buffer messages it receives from supplier handler threads. Monitor objects can help prevent the entire Gateway process from blocking whenever consumer handler threads encounter flow control on TCP connections to their remote consumers.

The following C++ class defines the interface for a message queue monitor object:

```
class Message_Queue
{
public:
  enum {
    MAX_MESSAGES = /* ... */;
  };

  // The constructor defines the maximum number
  // of messages in the queue.  This determines
  // when the queue is 'full.'
  Message_Queue (size_t max_messages
                 = MAX_MESSAGES);

  // = Message queue synchronized methods.

  // Put the <Message> at the tail of the queue.
  // If the queue is full, block until the queue
  // is not full.
  void put (const Message &msg);

  // Get the <Message> at the head of the queue.
  // If the queue is empty, block until the queue
  // is not empty.
  Message get (void);

  // True if the queue is full, else false.
  // Does not block.
  bool empty (void) const;

  // True if the queue is empty, else false.
  // Does not block.
  bool full (void) const;

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

The `Message_Queue` monitor object interface exports four synchronized methods. The `empty` and `full` methods are predicates that clients can use to distinguish three internal states: (1) empty, (2) full, and (3) neither empty nor full. The `put` and `get` methods enqueue and dequeue `Messages` into and from the queue, respectively, and will block if the queue is full or empty.

2. Define the method object's implementation methods: A monitor object often contains implementation methods that simplify its interface methods. This separation of concerns helps to decouple synchronization and scheduling logic from monitor object functionality, as well as avoid intra-object deadlock and unnecessary locking overhead.

The following conventions based on the Thread-safe Interface idiom [6] can be used to structure the separation of concerns between interface and implementation methods:

- Interface methods only acquire/release monitor locks and wait/notify certain monitor conditions, and then forward to implementation methods that perform the monitor object's functionality.
- Implementation methods only perform work when called by interface methods, *i.e.*, they do not acquire/release the monitor lock or wait/notify monitor conditions explicitly. Moreover, to avoid intra-object method deadlock or unnecessary synchronization overhead, implementation methods should not call any synchronized methods defined in their monitor object's interface.

In our Gateway example, the `Message_Queue` class defines four implementation methods: `put_i`, `get_i`, `empty_i`, and `full_i`, corresponding to the synchronized method interface. The signatures of these methods are shown below:

```
class Message_Queue
{
public:
  // ... See above ....

private:
  // = Private helper methods (non-synchronized
  //   and do not block).

  // Put the <Message> at the tail of the queue.
  void put_i (const Message &msg);

  // Get the <Message> at the head of the queue.
  Message get_i (void);

  // True if the queue is full, else false.
  // Assumes locks are held.
  bool empty_i (void) const;

  // True if the queue is empty, else false.
  // Assumes locks are held.
  bool full_i (void) const;

  // ...
```

The implementation methods are typically not synchronized, nor do they block, in accordance with the Thread-safe Interface idiom [6] outlined above.

3. Define the method object's internal state: A monitor object contains data members that define its internal state. In addition, a monitor object contains a monitor lock that serializes the execution of its synchronized methods and one or more monitor conditions used to schedule synchronized method execution within a monitor object. There is typically a separate monitor condition for each type of situation where synchronized methods must suspend themselves and/or resume other threads whose synchronized methods are suspended.

A monitor lock can be implemented using a *mutex*. A mutex makes collaborating threads wait while the thread holding the mutex executes code in a critical section. Monitor conditions can be implemented using *condition variables* [4]. Unlike a mutex, a condition variable is used by a thread to make *itself* wait until an arbitrarily complex condition expression involving shared data attains a particular state.

A condition variable is always used in conjunction with a mutex, which the client thread must acquire before evaluating the condition expression. If the condition expression is false, the client atomically suspends itself on the condition variable and releases the mutex so that other threads can change the shared data. When a cooperating thread changes this data, it can notify the condition variable, which atomically resumes a thread that had previously suspended itself on the condition variable and acquires its mutex again.

With its mutex held, the newly resumed thread then re-evaluates its condition expression. If the shared data has attained the desired state the thread continues. Otherwise, it suspends itself on the condition variable again until it's resumed. This process can repeat until the condition expression becomes true.

In general, a condition variable is more appropriate than a mutex for situations involving complex condition expressions or scheduling behaviors. For instance, condition variables can be used to implement thread-safe message queues. In this use case, a pair of condition variables can cooperatively block supplier threads when a message queue is full and block consumer threads when the queue is empty.

In our Gateway example, the `Message_Queue` defines its internal state as illustrated below:

```
class Message_Queue
{
  // ... See above ....

private:

  // Internal Queue representation.
  ...

  // Current number of <Message>s in the queue.
  size_t message_count_;

  // The maximum number <Message>s that can be
  // in a queue before it's considered 'full.'
  size_t max_messages_;

  // = Mechanisms required to implement the
  // monitor object's synchronization policies.

  // Mutex that protect the queue's internal state
  // from race conditions during concurrent access.
  mutable Thread_Mutex monitor_lock_;

  // Condition variable used to make synchronized
  // method threads wait until the queue is no
  // longer empty.
  Thread_Condition not_empty_;

  // Condition variable used to make synchronized
  // method threads wait until the queue is
  // no longer full.
  Thread_Condition not_full_;
};
```

A `Message_Queue` monitor object defines three types of internal state:

• **Queue representation data members:** These data members define the internal queue representation. This representation stores the contents of the queue in a circular array or linked list, along with bookkeeping information needed to determine whether the queue is empty, full, or neither. The internal queue representation is accessed and manipulated only by the `get_i`, `put_i`, `empty_i`, and `full_i` implementation methods.

• **Monitor lock data member:** The monitor lock is used by a `Message_Queue`'s synchronized methods to serialize their access to a monitor object. The monitor lock is implemented using the `Thread_Mutex` defined in the Wrapper Facade pattern [3]. This class provides a platform-independent mutex API.

• **Monitor condition data members:** The monitor conditions that the `put` and `get` synchronized methods use to suspend and resume themselves when a `Message_Queue` transitions between its full and empty boundary conditions, respectively. These monitor conditions are implemented using the `Thread_Condition` wrapper facade defined below:

```
class Thread_Condition
{
public:
  // Initialize the condition variable and
  // associate it with the <mutex_>.
  Thread_Condition (const Thread_Mutex &m)

  // Implicitly destroy the condition variable.
  ~Thread_Condition (void);

  // Wait for the <Thread_Condition> to be,
  // notified or until <timeout> has elapsed.
  // If <timeout> == 0 wait indefinitely.
  int wait (Time_Value *timeout = 0) const;

  // Notify one thread waiting on the
  // <Thread_Condition>.
  int notify (void) const;

  // Notify *all* threads waiting on
  // the <Thread_Condition>.
  int notify_all (void) const;

private:
#if defined (_POSIX_PTHREAD_SEMANTICS)
  pthread_cond_t cond_;
#else
  // Condition variable emulations.
#endif /* _POSIX_PTHREAD_SEMANTICS */

  // Reference to mutex lock.
  const Thread_Mutex &mutex_;
};
```

The constructor initializes the condition variable and associates it with the `Thread_Mutex` passed as a parameter. The destructor destroys the condition variable, which release any resources allocated by the constructor. Note that the `mutex_` is not owned by the `Thread_Condition`, so it is not destroyed in the destructor.

When called by a client thread, the wait method (1) atomically releases the associated mutex_ and (2) suspends itself for up to timeout amount of time waiting for the Thread_Condition object to be notified by another thread. The notify method resumes one thread waiting on a Thread_Condition and the notify_all method notifies *all* threads that are currently waiting on a Thread_Condition. The mutex_ lock is reacquired by the wait method before it returns to its client thread, *e.g.*, either because the condition variable was notified or because its timeout expired.

4. Implement all the monitor object's methods and data members: The final step involves implementing all the monitor object methods and internal state defined above. These steps can be further decomposed as follows:

• **Initialize the data members:** This substep initializes object-specific data members, as well as the monitor lock and any monitor conditions.

For instance, the constructor of Message_Queue creates an empty message queue and initializes the monitor conditions, not_empty_ and not_full_, as shown below.

```
Message_Queue::Message_Queue (size_t max_messages)
  : not_full_ (monitor_lock_),
    not_empty_ (monitor_lock_),
    max_messages_ (max_messages),
    message_count_ (0)
{
  // ...
}
```

In this example, now how both monitor conditions share the same monitor_lock_. This design ensures that Message_Queue state, such as the message_count_, is serialized properly to prevent race conditions when multiple threads put and get messages into a queue simultaneously.

• **Apply the Thread-safe Interface idiom:** In this substep, the interface and implementation methods are implementing according to the Thread-safe Interface idiom.

For instance, the following Message_Queue methods check if a queue is *empty*, *i.e.,* contains no Messages at all, or *full i.e.,* contains more than max_messages_ in it. We show the interface methods first:

```
bool
Message_Queue::empty (void) const
{
  Guard<Thread_Mutex> guard (monitor_lock_);
  return empty_i ();
}

bool
Message_Queue::full (void) const
{
  Guard<Thread_Mutex> guard (monitor_lock_);
  return full_i ();
}
```

These methods illustrate a simple example of the Thread-safe Interface idiom outlined above. They use the Scoped Locking idiom [6] to acquire/release the monitor lock and then immediately forward to the corresponding implementation method. As shown next, these methods assume the monitor_lock_ is held and simply check for the boundary conditions in the queue:

```
bool
Message_Queue::empty_i (void) const
{
  return message_count_ <= 0;
}

bool
Message_Queue::full_i (void) const
{
  return message_count_ > max_messages_;
}
```

The put method inserts a new Message at the tail of a queue. It is a synchronized method that illustrates a more sophisticated use of the Thread-safe Interface idiom:

```
void
Message_Queue::put (const Message &msg)
{
  // Use the Scoped Locking idiom to
  // acquire/release the <monitor_lock_> upon
  // entry/exit to the synchronized method.
  Guard<Thread_Mutex> guard (monitor_lock_);

  // Wait while the queue is full.

  while (full_i ()) {
    // Release <monitor_lock_> and suspend our
    // thread waiting for space to become available
    // in the queue.  The <monitor_lock_> is
    // reacquired automatically when <wait> returns.
    not_full_.wait ();
  }

  // Enqueue the <Message> at the tail of
  // the queue and update <message_count_>.
  put_i (new_item);

  // Notify any thread waiting in <get> that
  // the queue has at least one <Message>.
  not_empty_.notify ();

  // Destructor of <guard> releases <monitor_lock_>.
}
```

Note how this synchronized method only performs the synchronization and scheduling logic needed to serialize access to the monitor object and wait while the queue is full, respectively. Once there's room in the queue, it forwards to the put_i method, which inserts the message into the queue and updates the bookkeeping information. Moreover, the put_i need not be synchronized because the put method never calls it without first acquiring the monitor_lock_. Likewise, the put_i method need not check to see if the queue is full because it is never called as long as full_i returns true.

The get method removes the Message from the front of a queue and returns it to the caller.

```
Message
Message_Queue::get (void)
{
  // Use the Scoped Locking idiom to
  // acquire/release the <monitor_lock_> upon
  // entry/exit to the synchronized method.
  Guard<Thread_Mutex> guard (monitor_lock_);

  // Wait while the queue is empty.

  while (empty_i ()) {
    // Release <monitor_lock_> and wait for a new
```

```
    // <Message> to be placed in the queue.  The
    // <monitor_lock_> is reacquired automatically
    // when <wait> returns.
    not_empty_.wait ();
  }

  // Dequeue the first <Message> in the queue
  // and update the <message_count_>.
  Message m = get_i ();

  // Notify any thread waiting in <put> that the
  // queue has room for at least one <Message>.
  not_full_.notify ();

  return m;
  // Destructor of <guard> releases <monitor_lock_>.
}
```

As before, note how the `get` synchronized method focuses on the synchronization and scheduling logic, while forwarding the actual dequeueing operation to the `get_i` method.

10 Example Resolved

The Gateway application can use the Monitor Object pattern to implement a thread-safe message queue that decouples supplier handler and consumer handler threads so they run concurrently and block independently. Embedding and automating synchronization inside message queue monitor objects protects their internal state from corruption and shields clients from low-level synchronization concerns.

Internally, the Gateway contains `Supplier_Handler` and `Consumer_Handler` objects that act as local proxies [7, 8] for remote suppliers and consumers, respectively. Each `Consumer_Handler` contains a `Message_Queue` object implemented using the Monitor Object pattern as described in the *Implementation* section. The `Consumer_Handler` class is defined as follows:

```
class Consumer_Handler
{
public:
  Consumer_Handler (void);

  // Put the message into the queue
  // monitor object, blocking until
  // there's room in the queue.
  void put (const Message &msg) {
    message_queue_.put (msg);
  }

private:
  // Message queue implemented as a
  // monitor object.
  Message_Queue message_queue_;

  // Connection to the remote consumer.
  SOCK_Stream connection_;

  // Entry point into a new consumer
  // handler thread.
  static void *svc_run (void *arg);
};
```

As shown in Figure 3, each `Supplier_Handler` runs in its own thread, receive messages from its remote supplier, and routes the messages to their remote consumers. Routing is performed by inspecting an address field in each message,

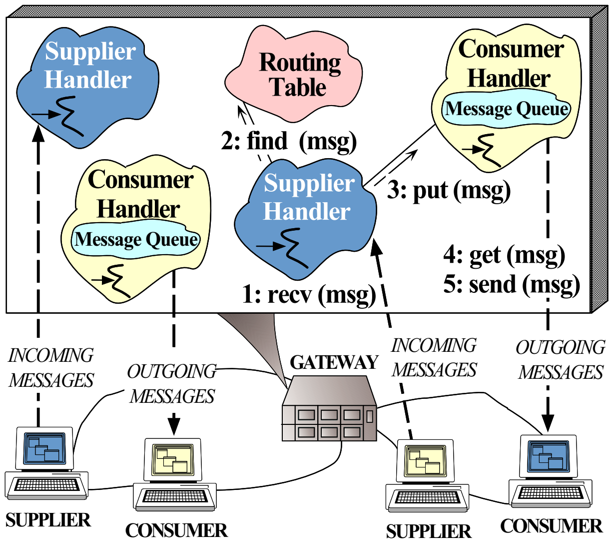


Figure 3: Implementing the Communication Gateway Using the Monitor Object Pattern

which is used as a key into a routing table that maps keys to `Consumer_Handlers`. Each `Consumer_Handler` is responsible for receiving messages from suppliers via its `put` method and storing each message its `Message_Queue` monitor object, as follows:

```
Supplier_Handler::route_message (const Message &msg)
{
  // Locate the appropriate consumer based on the
  // address information in the <Message>.
  Consumer_Handler *ch =
    routing_table_.find (msg.address ());

  // Put the <Message> into the <Consumer Handler>,
  // which will store it in its <Message Queue>
  // monitor object.
  ch->put (msg);
};
```

Each `Consumer_Handler` spawns a separate thread of control in its constructor to process the messages placed into its message queue, as follows:

```
Consumer_Handler::Consumer_Handler (void)
{
  // Spawn a separate thread to get messages
  // from the message queue and send them to
  // the remote consumer via TCP.
  Thread_Manager::instance ()->spawn (svc_run,
                                      this);
}
```

Each `Consumer_Handler` thread executes the `svc_run` method, which gets the messages placed into the queue by `Supplier_Handler` threads and sends them over its TCP connection to the remote consumer, as follows:

```
void *
Consumer_Handler::svc_run (void *args)
{
  Consumer_Handler *this_obj =
    reinterpret_cast<Consumer_Handler *> (args);
```

```
  for (;;) {
    // This thread blocks on <get> until the
    // next <Message> is available.
    Message msg =
      this_obj->message_queue_.get ();

    // Transmit message to the consumer.
    this_obj->connection_.send (msg,
                                msg.length ());
  }
}
```

The Message_Queue is implemented as a monitor object. Therefore, the send operation on the connection_ can block in a Consumer_Handler without affecting the quality of service of other Consumer_Handlers or Supplier_Handlers.

11 Variants

The following are variations of the Monitor Object pattern.

Timed synchronized method invocations: Many applications can benefit from timed synchronized method invocations. Timed invocations enable clients to bound the amount of time they are willing to wait for a synchronized method to enter its monitor object's critical section.

The Message_Queue monitor object interface defined earlier can be modified to support timed synchronized method invocations, as follows:

```
class Message_Queue
{
public:
  // = Message queue synchronized methods.

  // Put the <Message> at the tail of the queue.
  // If the queue is full, block until the queue
  // is not full.  If <timeout> is 0 then block
  // until the <Message> is inserted into the queue.
  // Otherwise, if <timeout> expires before the
  // <Message> is enqueued, the <Timedout> exception
  // is thrown.
  void put (const Message &msg,
            Time_Value *timeout = 0)
       throw (Timedout);

  // Get the <Message> at the head of the queue.
  // If the queue is empty, block until the queue
  // is not empty.  If <timeout> is 0 then block
  // until the <Message> is inserted into the queue.
  // Otherwise, if <timeout> expires before the
  // <Message> is enqueued, the <Timedout> exception
  // is thrown.
  Message get (Time_Value *timeout = 0)
    throw (Timedout);

  // ...
};
```

If timeout is 0 then both get and put will block indefinitely until a Message is either removed or inserted into a Message_Queue monitor object, respectively. Otherwise, if the timeout period expires, the Timedout exception is thrown and the client must be prepared to handle this exception.

The following illustrates how the put method can be implemented using the timed wait feature of the Thread_Condition condition variable wrapper outlined in the *Implementation* section:

```
void
Message_Queue::put (const Message &msg,
                    Time_Value *timeout)
  throw (Timedout)
{
  // ... Same as before ...

  // Wait while the queue is full.

  while (full_i ()) {
    // Release <monitor_lock_> and suspend our
    // thread waiting for space to become available
    // in the queue or for <timeout> to elapse.
    // The <monitor_lock_> is reacquired automatically
    // when <wait> returns, regardless of whether
    // a timeout occurred or not.
    if (not_full_.wait (timeout) == -1
        && errno = ETIMEDOUT)
      throw Timedout ();
  }

  // ... Same as before ...
}
```

Strategized locking: The Strategized Locking pattern can be applied to make a monitor object more flexible and reusable.

For instance, the following template class parameterizes the synchronization aspects of a Message_Queue:

```
template <class SYNCH_STRATEGY>
class Message_Queue
{
  // ...

private:
  typename SYNCH_STRATEGY::MUTEX monitor_lock_;
  typename SYNCH_STRATEGY::CONDITION not_empty_;
  typename SYNCH_STRATEGY::CONDITION not_full_;
  // ...
};
```

Each synchronized method is then modified as shown by the following empty method:

```
template <class SYNCH_STRATEGY> bool
Message_Queue<SYNCH_STRATEGY::empty (void) const
{
  Guard<SYNCH_STRATEGY::MUTEX> guard (monitor_lock_);
  return empty_i ();
}
```

To parameterize the synchronization aspects associated with a Message_Queue, we can define a pair of classes, MT_SYNCH and NULL_SYNCH, that typedef the appropriate C++ traits, as follows:

```
class MT_SYNCH {
public:
  // Synchronization traits.
  typedef Thread_Mutex MUTEX;
  typedef Thread_Condition CONDITION;
};

class NULL_SYNCH {
```

```
  // Synchronization traits.
  typedef Null_Mutex MUTEX;
  typedef Null_Thread_Condition CONDITION;
};
```

Thus, to define a thread-safe `Message_Queue`, we just parameterize it with the `MT_SYNCH` strategy, as follows:

```
Message_Queue<MT_SYNCH> message_queue;
```

Likewise, to create a non-thread-safe `Message_Queue`, we can simply parameterize it with the following `NULL_SYNCH` strategy.

```
Message_Queue<NULL_SYNCH> message_queue;
```

Note that when using the Strategized Locking pattern in C++, it may not be possible for the component class to know what type of synchronization strategy will be configured for a particular use case. Therefore, it is important to apply the Thread-safe Interface idiom to ensure that intra-object method calls, such as `put` calling `full_` and `put_i`, avoid self-deadlock and/or minimize recursive locking overhead.

12 Known Uses

The following are some known uses of the Monitor Object pattern:

Dijkstra/Hoare monitors: Dijkstra [9] and Hoare [5] defined programming language features called *monitors* to encapsulate service functions and their internal variables into thread-safe modules. To prevent race conditions, a monitor contains a lock that allows only one function at a time to be active within the monitor. Functions that want to temporarily leave the monitor can block on a condition variable. It is the responsibility of the programming language compiler to generate run-time code that implements and manages the monitor lock and condition variables.

Java Objects: The main synchronization mechanism in Java is based on Dijkstra/Hoare-style monitors. Each Java object is implicitly a monitor that internally contains a monitor lock and a single monitor condition. Java's monitors are relatively simple, *i.e.*, they allow threads to (1) implicitly serialize their execution via method-call interfaces and (2) to coordinate their activities via explicit `wait`, `notify`, and `notifyAll` operations.

ACE Gateway: The example from Section 10 is based on a communication Gateway [10] from the ACE framework. The `Message Queues` used by `Consumer Handlers` in the Gateway are reusable ACE components implemented as monitor objects. The Monitor Object pattern is used in the ACE Gateway to simplify concurrent programming and improve performance on multi-processors.

13 Consequences

The Monitor Object pattern provides the following **benefits**:

Simplify synchronization of methods invoked concurrently on an object: Clients need not be concerned with concurrency control when invoking methods on a monitor object. If a programming language doesn't support monitor objects as a language feature, developers can use idioms like Scoped Locking [6] to simplify and automate the acquisition and release of monitor locks that serializes access to internal monitor object methods and state.

Synchronized methods can cooperatively schedule their order of execution: Synchronized methods use their monitor conditions to determine the circumstances under which they should suspend or resume their execution. For instance, methods can suspend themselves and wait to be notified when arbitrarily complex conditions occur *without* using inefficient polling. This feature makes it possible for monitor objects to cooperatively schedule their methods in separate threads.

However, the Monitor Object pattern has the following **liabilities**:

Tightly coupling between object functionality and synchronization mechanisms: It is usually straightforward to decouple an active object's functionality from its synchronization policies because it has a separate scheduler. In contrast, a monitor object's synchronization and scheduling logic is often closely coupled with its methods' functionality. Although this makes monitor objects more efficient than active objects, it may be hard to change their synchronization policies or mechanisms without directly changing the monitor object's method implementations. One way to reduce the coupling of synchronization and functionality in monitor objects is to use Aspect-Oriented Programming, as described in Thread-Safe Interface idiom and Strategized Locking pattern [6].

Nested monitor lockout: This problem can occur when a monitor object is nested within another monitor object. For instance, consider the following two Java classes:

```
class Inner {
  protected boolean cond_ = false;

  public synchronized void awaitCondition () {
    while (!cond)
      try { wait (); }
      catch (InterruptedException e) {}
      // Any other code.
  }

  public synchronized
    void notifyCondition (boolean c) {
    cond_ = c;
    notifyAll ();
  }
}

class Outer {
  protected Inner inner_ =
    new Inner ();

  public synchronized void process () {
    inner_.awaitCondition ();
  }
```

```
  public synchronized
    void set (boolean c) {
    inner_.notifyCondition (c);
  }
}
```

The code above illustrates the canonical form of the the nested monitor lockout problem in Java. When a Java thread blocks in the monitor's wait queue, all its locks are held *except* the lock of the object placed in the queue. Consider what would happen if thread T_1 made a call to `Outer.process` and as a result blocked in the `wait` call in `Inner.awaitCondition`. In Java, the `Inner` and `Outer` classes do not share their monitor locks. Thus, the `awaitCondition` call would release the `Inner` monitor, while retaining the `Outer` monitor. However, another thread, T_2 cannot acquire the `Outer` monitor because it is locked by the synchronized `process` method. As a result, the `Outer.set` condition cannot become true and T_1 will continue to block in `wait` forever. Techniques for avoiding nested monitor lockout in Java are described in [11, 12].

14 See Also

The Monitor Object pattern has several properties in common with the Active Object pattern [2]. For instance, both patterns can be used to synchronize and schedule methods invoked concurrently on an object. One difference is that an active object executes its methods in a different thread than its client(s), whereas a monitor object executes its methods in its client threads. As a result, active objects can perform more sophisticated, albeit more expensive, scheduling to determine the order in which their methods execute. Another difference is that monitor objects typically couple their synchronization logic more closely with their methods' functionality. In contrast, it is easier to decouple an active object's functionality from its synchronization policies because it has a separate scheduler.

For example, it is instructive to compare the Monitor Object solution in Section 10 with the solution presented in the Active Object [2] pattern. Both solutions have similar overall application architectures. In particular, the `Supplier_Handler` and `Consumer_Handler` implementations are almost identical. The primary difference is that the `Message_Queue` itself is easier to program and is more efficient when it's implemented using the Monitor Object pattern rather than the Active Object pattern.

If a more sophisticated queueing strategy was necessary, however, the Active Object pattern might be more appropriate. Likewise, because active objects execute in different threads than their clients, there are use cases where active objects can improve overall application concurrency by executing multiple operations asynchronously. When these operations are complete, clients can obtain their results via futures.

Acknowledgements

References

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

[2] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[3] D. C. Schmidt, "Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes," *C++ Report*, vol. 11, February 1999.

[4] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[5] C. A. R. Hoare, "Monitors: An Operating System Structuring Mechanism," *Communications of the ACM*, vol. 17, Oct. 1974.

[6] D. C. Schmidt, "Strategized Locking, Thread-safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components," *C++ Report*, vol. 11, Sept. 1999.

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

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

[9] E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages* (F. Genuys, ed.), Reading, MA: Academic Press, 1968.

[10] 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.

[11] D. Lea, *Concurrent Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley, 1996.

[12] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.