Better Monitors for Java

Theodore S. Norvell
Electrical and Computer Engineering
Memorial University

First published by JavaWorld.com at http://www.javaworld.com,
copyright International Data Group, October 2007.

It is all too easy to write unsafe multithreaded Java code, in part because the language's notify() and wait() methods can be difficult to use correctly. In 1974, Tony Hoare proposed the concept of *monitors* for designing and reasoning about objects that are shared between multiple threads. The key property of these "Hoare-style" monitors is that threads can wait until some specific assertion about the monitor's state is true and will be guaranteed that the assertion is still true after the thread has been awakened.

In this article, I describe a Java class library that supports the monitor concept much as Hoare originally proposed it. As I will demonstrate, object-oriented technology makes it possible to improve on Hoare's concept by making assertions — which are traditionally stated as comments and then repeated in *if* statements and *assert* statements — executable objects. This approach reduces redundant coding and automates assertions checking during execution.

I'll start by explaining why multithreaded programming is as unavoidable as it is challenging, then introduce you to the use of exclusive access, conditions, and assertions in Hoare-style monitors. See the Resources section to download the Java-based monitors package, which you are free to use.

1 The case for Hoare-style monitors

Three factors contribute to the ubiquity of multithreaded software, and thus the need for thread monitors:

- First, to create responsive interactive software, we need to perform tasks that take more than a few tens of milliseconds on threads other than the GUI event thread.

- Second is the drive to make the most of today's multiprocessor, multicore, and simultaneous multithreading (hyper-threading) hardware. Computationally intensive applications must multiple threads to use more than a fraction of the capability of that hardware.
- Third is distributed computing, which demands that applications deal with multiple requests simultaneously; for example, RMI invocations each run on their own thread.

Coordinating the activities of two or more threads can be very tricky and can be a source of latent defects in multithreaded code. Testing is ineffective because tests may not reveal race conditions. As software engineers we must equip ourselves with the intellectual and software tools to design correct multithreaded applications.

1.1 Monitoring multithreaded access

In the nineteen-seventies Edsger Dijkstra [1], Per Brinch Hansen [2], and C.A.R. Hoare [3] developed the idea of a *monitor*, an object that would protect data by enforcing single threaded access — only one thread at a time would be allowed to execute the code within the monitor. A monitor would be an island of single-threaded calm in a turbulent sea of multithreadedness. If threads could confine their interactions to be within the monitors then there would be a much better chance that they wouldn't interfere with each other in unanticipated ways.

A number of variants on the idea of monitors have been proposed and implemented in various programming languages such as Concurrent Pascal, Mesa, Concurrent Euclid, Turing, and more recently Java. The idea of single-threaded access to a monitor is straightforward and the same in all these variants; where they differ is in how threads wait for the state of the monitor to change, and how they communicate to each other that the state has changed. Of all these approaches, I've always found Hoare's version the easiest to use. In Hoare's version, occupancy of the monitor is transferred directly from a signalling to a signalled thread: the state of the monitor object can not change in between the signalling thread leaving and the signalled thread arriving.

1.2 A Java-based implementation of Hoare-style monitors

For the rest of this article I'll focus on a Java package I've developed that implements Hoare-style monitors. In addition to being a straightforward Java-based implementation of Tony Hoare's original concept, the monitor package has one novel feature: it allows all assertions used in designing a monitor to be expressed as part of the code and checked at runtime.

To get started, table 1 summarizes the classes in the monitor package. See the Resources section for the source code for the monitor package.

public abstract class AbstractMonitor	Abstract base class for monitor
protected AbstractMonitor()	Constructor
protected boolean invariant()	Hook method for invariant
protected void enter()	Obtain occupancy
protected void leave()	Relinquish occupancy
protected Condition makeCondition(Assertion p)	Make a condition object
protected Condition makeCondition()	Make a condition with an always true assertion
public class Monitor	Monitor objects for delegation.
public Monitor(Assertion invariant)	Construct a monitor with a given invariant
public void enter()	Obtain occupancy
public void leave()	Relinquish occupancy
public Condition makeCondition(Assertion p)	Make a condition object
public Condition makeCondition()	Make a condition with an always true assertion
public class Condition	Condition objects
public void await()	Wait for a P_c to become true
public void conditionalAwait()	If P_c is not true, wait until it is
public void signal()	Signal that P_c is true
public void conditionalSignal()	If P_c is true, signal so
public void signalAndLeave()	Signal that P_c is true and leave the monitor
public void conditionalSignalAndLeave()	If P_c is true, signal so and leave the monitor
public int count()	The number of threads waiting
public boolean isEmpty()	Is the number of threads waiting 0?
public abstract class Assertion	Executable assertion objects
public abstract boolean isTrue()	Hook method

Table 1: Summary of interfaces

2 Monitor concepts: Exclusive access

A *monitor* is an object that only allows exclusive access to its data; that is, data can be accessed by only one thread at a time. In my monitor package, exclusive access is accomplished by inheriting from class AbstractMonitor and calling inherited method enter() to enter the monitor and inherited method leave() to leave it.

Between returning from enter() and invoking leave() a thread is said to *occupy* the monitor. At most one thread can occupy the monitor at any time; a thread that invokes enter() while the monitor is occupied must wait (on an *entry queue* associated with the monitor) until the monitor becomes unoccupied. When a thread calls leave(), one thread waiting on the entry queue — if there is such a thread — is selected and allowed to occupy the monitor.

Consider the class in Example 1. Without the exclusive access provided by the calls to enter() and leave() a client might find that the time is 25:60:60 (which should be impossible) or might find the time to be 13:59:59 when it is really 13:00:00.

Note that this example could equally well be implemented using Java's **synchronized** keyword. The only advantage to using the monitor package for this example is that the class invariant, embodied as the overridden method invariant(), is checked as part of the entering and leaving process. The invariant describes the set of states the monitor may be in when the monitor is not occupied. If the invariant is found to be false on either a return from enter() or a call to leave(), an Error exception will be thrown. In such a simple class, checking the class invariant is probably not that useful. However, in more complex classes, checks of monitor invariants can provide a valuable indication of a bug during testing or even when the product is deployed.

I marked the fields of the example class as **volatile** so that the compiler will not assume that only one thread accesses these fields. Generally, to be on the safe side, fields of a monitor should be marked either as **final** or **volatile**.

Since Java does not support multiple inheritance, it is sometimes inconvenient to inherit from AbstractMonitor. In this case one can use a private field of class Monitor. Monitor is just like AbstractMonitor, except its methods are public rather than protected.

2.1 Contacts for enter() and leave()

The enter() and leave() methods are well described by contracts, i.e., their preconditions and postconditions. Here I refers to the monitor's invariant.

m.enter()	
Require:	this thread does not occupy the monitor
Ensure:	I && this thread alone occupies the monitor

```
import monitor.* ;

public class TimeOfDay extends AbstractMonitor{
    private volatile int hr = 0, min = 0, sec = 0 ;

    public void tick() {
        enter() ;
            sec += 1 ; min += sec/60 ; hr += min/60 ;
            sec = sec % 60 ; min = min % 60 ; hr = hr % 24 ;
        leave() ; }

    public void get( int[] time ) {
        enter() ;
            time[0] = sec ; time[1] = min ; time[2] = hr ;
        leave() ; }

    @Override
    protected boolean invariant() {
        return 0 <= sec && sec < 60
            && 0 <= min && min < 60
            && 0 <= hr && hr < 24 ; }
}
```

Example 1: A class with exclusive access

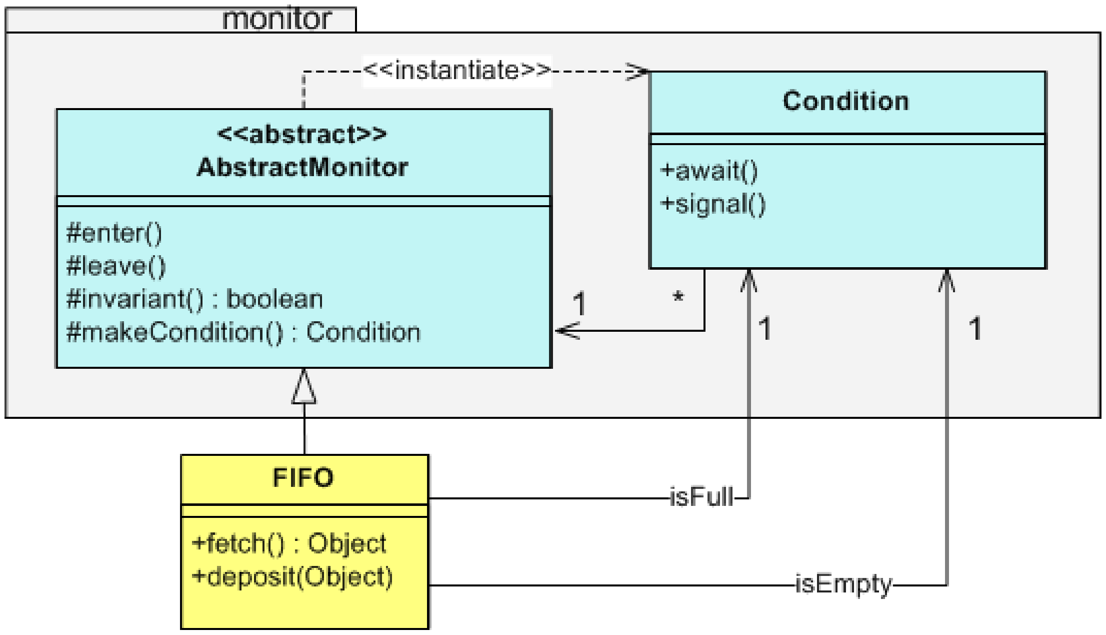

Figure 1: A UML class diagram for example 2.

m.leave()	
Require:	I && this thread alone occupies the monitor
Ensure:	this thread does not occupy the monitor

3 Conditions

The real power of Hoare-style monitors becomes evident when a thread may have to wait for something before completing its work in the monitor.

3.1 Condition Objects

A classic example is a bounded FIFO (First In, First Out) queue. A thread that is fetching from the queue must wait until the queue is not empty; a thread that is depositing data into the queue, must wait until there is room for data in the queue. Example 2 shows a FIFO queue with a capacity of 10 objects, implemented in Java using my monitor package. Figure 1 is a UML class diagram for the FIFO queue in Listing 2.

Example 2 shows a FIFO queue with a capacity of ten objects, implemented in Java using my monitor package. Figure 1 shows a UML class diagram for the example.

```
import monitor.* ;

public class FIFO extends AbstractMonitor {
    private final int capacity = 10 ;

    private final Object[] queue = new Object[ capacity ] ;

    private volatile int count = 0, front = 0 ;

    /** Awaiting ensures: count < capacity */
    private final Condition notFull = makeCondition() ;

    /** Awaiting ensures: count > 0 */
    private final Condition notEmpty = makeCondition() ;

    public Object fetch() {
        enter() ;
            if( ! (count > 0 ) ) { notEmpty.await() ; assert count > 0 ; }
            Object value = queue[ front ] ;
            --count;
            front = (front + 1) % capacity ;
            assert count < capacity ; notFull.signal() ;
        leave() ;
        return value ; }

    public void deposit( Object value ) {
        enter() ;
            if( ! ( count < capacity ) ) {
                notFull.await() ; assert count < capacity ; }
            queue[ (front + count) % capacity ] = value ; ++count ;
            assert count > 0 ; notEmpty.signal() ;
        leave() ; }

    @Override
    protected boolean invariant() {
        return 0<=count && count<=capacity && 0<=front && front<capacity ; }
}
```

Example 2: A fifo queue

The number of objects present in the queue is count. A fetching thread may have to wait until count > 0, while a depositing thread may have to wait until count $<$ capacity. Each of these assertions about the state is represented by a *condition object*. The two condition objects are referenced by two private fields and are created by calls to method makeCondition() inherited from AbstractMonitor.

```
/** Awaiting ensures: count < capacity */
private final Condition notFull = makeCondition() ;

/** Awaiting ensures: count > 0 */
private final Condition notEmpty = makeCondition() ;
```

Each condition object c is associated with an assertion P_c. In the example P_{notFull} is count<capacity, while P_{notEmpty} is count>0

When a thread needs to wait for an assertion P_c to become true, it invokes c.await(). The effect of c.await() is that the thread that calls it leaves the monitor and waits on a queue associated with the condition object. While the thread is waiting, it does not occupy the monitor, and so another thread can enter and possibly make the assertion become true. Since the thread that invokes await() leaves the monitor unoccupied, the class invariant should be true when await() is invoked, and indeed await() checks the invariant.

A thread that makes an assertion P_c true may (and usually should) transfer occupancy to some waiting thread, if there is one. It can do this by invoking the method c.signal(). If one or more threads are waiting on that condition object, one waiting thread is selected and it enters the monitor. As the selected thread enters the monitor, the signalling thread leaves the monitor to wait on the monitor's entry queue. At no point in time, during this exchange, is the monitor unoccupied; thus the waiting thread will find the monitor in the same state it was in when the signalling thread called signal(). Only when the monitor is once again empty does the signalling thread get to reoccupy the monitor and return from signal().

A flash animation at

`www.engr.mun.ca/~theo/Misc/monitors/monitor-movie.swf`

illustrates how the threads use enter(), leave(), await(), and signal().

3.2 Conditions and design-by-contract

If we follow the convention that the assertion P_c is true whenever c.signal() is called, then P_c must be true when c.await() returns. This is sort of a *contract* that the implementor of the monitor makes with his or herself. Indeed signal() and await() are best understood in terms of contracts, based on preconditions and postconditions.

3.2.1 A contract for wait()

The precondition of c.await(), as we've seen, is the monitor's invariant, which I'll call I. The thread only returns from c.await() when another thread calls c.signal(). From the precondition of c.signal(), as we will see, P_c must be true at that time. The contract for c.await() is

	c.await()
Require:	I && this thread alone occupies the monitor
Ensure:	P_c && this thread alone occupies the monitor

3.2.2 A contract for signal()

What about signal()? If there is a thread waiting on the condition's queue, then P_c must be true as a precondition. On return, as the monitor was just empty, the monitor invariant will be true as a postcondition. But what if there is no thread waiting? In that case, signal() does nothing and so, to ensure the invariant is true as a postcondition, it should be a precondition as well. You can express the contract using c.isEmpty(), which indicates whether any threads waiting on the condition.

	c.signal()
Require:	(!c.isEmpty() && P_c || c.isEmpty() && I) && this thread alone occupies the monitor
Ensure:	I && this thread alone occupies the monitor

This contract is a bit complicated; in practice we can almost always use one of two simpler preconditions instead; each implies the contract's precondition: one is

P_c&& I && this thread alone occupies the monitor

and the other is

!c.isEmpty() && P_c && this thread alone occupies the monitor

3.2.3 signalAndLeave()

Often, there is no need for a signalling thread to wait and later reoccupy the monitor. In this case, one can call signalAndLeave(). Upon awakening an awaiting thread, if there is one, this method returns without re-occupying the monitor. The contract for signalAndLeave() is

	c.signalAndLeave()
Require:	(!c.isEmpty() && P_c || c.isEmpty() && I) && this thread alone occupies the monitor
Ensure:	this thread does not occupy the monitor

There's one more wrinkle to these contracts: they only work if neither the invariant I nor the assertion P_c depend on the number of threads waiting on c. That's because the acts of starting to wait and ceasing to wait change this quantity. Howard [4] and Dahl [5] give correct contracts for the rare cases when I and P_c may depend on the number of waiting threads.

4 Assertion objects

Looking back at Example 2, you can see that each assertion appears four times, counting comments, assert statements, and if statements. For good reason software engineers abhor such duplication. Of course, you could encapsulate each duplicated assertion in a method, but then you would have to duplicate the call to the method. In my Java-based monitor package I've provided a better solution, wherein each assertion may be represented by an object.

Classes representing assertions are created by subclassing Assertion and overriding the abstract method isTrue(). Each condition object knows one Assertion object and checks the assertion upon any call to signal(), throwing an exception if the assertion is false. Based on this approach, you could change the FIFO queue in Example 2 by rewriting the declarations of the condition objects as follows:

```
private final Condition notFull = makeCondition(
    new Assertion() {
        public boolean isTrue() { return count < capacity ; } } ) ;

private final Condition notEmpty = makeCondition(
    new Assertion() {
        public boolean isTrue() { return count > 0 ; } } ) ;
```

You can also use assertion objects to replace if statements, such as those in fetch and deposit, with calls to conditionalAwait(). The rewritten deposit method is shown in Example 3; the fetch method can be rewritten similarly. The classes for the revised FIFO queue are shown in Figure 2. Similarly, there is a conditionalSignal() method.

In summary, the following checks are made automatically:

- The *invariant* is checked on the return from enter(), the call to leave(), on the call to await(), on the return from signal(), and, when there is no waiting thread, on the call to signalAndLeave().

```
public void deposit( Object value ) {
    enter() ;
        notFull.conditionalAwait() ;
        queue[ (front + count) % capacity ] = value ;
        count++ ;
    notEmpty.signalAndLeave() ; }
```

Example 3: The deposit method rewritten

- *Assertion objects* are checked on calls to signal() or signalAndLeave(), if there is a waiting thread, and on the return from await().

5 A voting example

As an interesting example of using Hoare-style monitors, let's look at how a number of threads can synchronize and exchange information. The specific problem is that a number of threads must vote on a boolean result. We'll use a simple majority, with ties going to false. Example 4 shows the monitor.

5.0.4 About the code

The constructor takes the number of threads involved in the elections. After construction each thread casts its vote and receives the majority using the castVoteAndWaitForResult method.

The election is over when votesFor+votesAgainst==N and so this assertion is $P_{\text{electionDone}}$. Interestingly this assertion contradicts the invariant. After casting its vote, a thread waits for the election to be over by conditionally waiting on electionDone. On returning from electionDone.conditionalAwait(), the thread can be sure that $P_{\text{electionDone}}$ is true. Thus it could leave the monitor and return with the result at that time.

This would cause two problems, however. First, only the last thread to vote would get the result; all others would wait forever. Second, there would be no way to reuse the monitor object for a second election. To solve these two problems I divided the work. Each thread awakes one waiting thread, until no threads are waiting on electionDone — I call this I call this technique "daisy chaining." The last thread on the chain finds there are no more threads to awake; it resets the state of the monitor to the original state; having done that, it has reestablished the monitor invariant and may leave the monitor unoccupied.

Note that the invariant is not true when signalAndLeave is called. Despite the fact that the invariant has not been reestablished, it is acceptable for the signalling threads to leave

```
import monitor.* ;

public class VoteMonitor extends AbstractMonitor {

    private final int N ;
    private volatile int votesFor = 0, votesAgainst = 0 ;

    private Condition electionDone = makeCondition( new Assertion() {
        public boolean isTrue() { return votesFor+votesAgainst == N ; } } ) ;

    public VoteMonitor(int N) { assert N>0 ; this.N = N ; }

    @Override
    protected boolean invariant() {
        return 0 <= votesFor && 0 <= votesAgainst && votesFor+votesAgainst < N ;
    }

    public boolean castVoteAndWaitForResult(boolean vote) {
        enter() ;

            if( vote ) votesFor++ ; else votesAgainst++ ;

            electionDone.conditionalAwait() ;
            // votesFor + votesAgainst == N

            boolean result = votesFor > votesAgainst ;

            if( ! electionDone.isEmpty() ) {
                electionDone.signalAndLeave() ; }
            else {
                // The last one woken up is the first one out and cleans up.
                votesFor = votesAgainst = 0 ;
                leave() ; }

        return result ;
    }
}
```

Example 4: The Vote Moitor

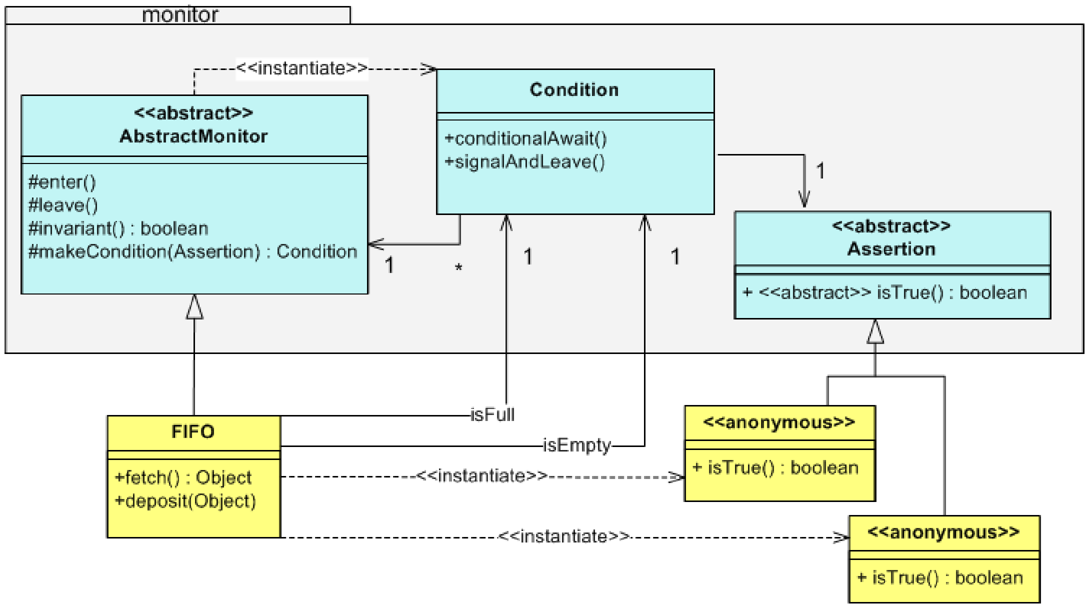

Figure 2: Classes for the revised FIFO queue.

the monitor, because they are not leaving it empty.

6 Comparison to Java's standard library

It wouldn't be right to present such a package without comparing it to Java's 'native' monitors built using the **synchronized** keyword and the wait() and notify() methods from the Object class.

In Java every object is associated with two queues: an entry queue and a wait queue. When a thread calls a method marked as **synchronized**, the thread will wait on the entry queue until no other thread occupies the object. On returning from a **synchronized** method the thread relinquishes occupancy. Threads unconditionally wait on the wait queue by calling the wait() method of the synchronized object. A call to notify() does not relinquish exclusive access, rather it simply moves some waiting thread, if there is one, from the object's wait queue to the object's entry queue. There is also a notifyAll() method that moves all threads from the wait queue to the entry queue.

Java's native monitors are similar to those in Mesa [6] and are sometimes said to use the "signal and continue" (SC) signalling discipline. By contrast, my monitor package gives the programmer a choice of "signal and wait" (SW) or "signal and leave" (SL).

Next I'll summarize the main differences between my monitor package and Java's wait()

and notify() approach.

Defensive programming: The monitor package provides support for defensive programming through the invariant() method and the assertion objects associated with each condition object. Java's notify() and wait() provide no assertion checking.

Multiple wait queues: The monitor package provides condition objects, each providing its own wait queue. Threads waiting for different assertions to become true wait on different queues. Using wait(), there is only one wait queue per object. Threads waiting for different assertions to become true must all wait on the same queue; thus a notify() call could awaken a thread that is waiting for the "wrong" assertion. The usual solution is to replace calls to notify() with calls to notifyAll() and to always put calls to wait() in a loop, so:

```
while( ! $P_c$ ) { wait() ; }
```

Flexible assertions: With Hoare-style monitors it is possible for a waiting thread to awaken in a state where the class invariant is not true. This gives the designer more flexibility in choosing the assertions that threads can wait for. When a thread returns from wait(), it does so from the entry queue and so the invariant must be true.

Seamless passing of occupancy: In the monitor package, following Hoare's suggestion, control is passed seamlessly from the signalling thread to an awaiting thread. You can be sure that, after returning from an await(), the assertion the thread has waited for is true. In contrast, the notify() and notifyAll() methods simply move a waiting thread back to the entry queue for the monitor. By the time the waiting thread actually becomes the occupant, the assertion for which it was waiting may no longer be true. This is a second reason for putting wait() calls in a loop. But such loops have a problem: an invocation of notify() that awakes a thread that then re-waits is effectively lost. As notify() calls can be lost, it can be difficult to ensure "sufficient signalling" — the property that any waiting thread is eventually awakened. With the monitor package, ensuring sufficient signalling requires only ensuring that each await() be matched by some signal() on the same condition object in the future.

7 Conclusion

Hoare-style monitors provide a sound programming method for coordinating the interactions of multiple threads. In the limited space of this article, I could only illustrate this point with very simple examples. As the complexity of the interactions increase, the benefit of using Hoare-style monitors increases.

In Hoare's original concept, a monitor's invariant and the assertions associated with condition variables were purely tools of design and documentation: the truth of the assertions can be proved and there is be no need to check them at runtime. In practice though, even careful designers make mistakes and, as software evolves, errors can creep in. Thus it is pragmatic to check assertions at runtime. The monitor package I've presented in this article automatically checks assertions and invariants, letting us move the assertions out of the comments and into the code.

Source code for the monitor package can be found at

`www.engr.mun.ca/~theo/Misc/monitors/monitors.html`

8 Acknowledgements

Thanks are due to Alain O'Dea for suggesting an improvement to the monitor package, to Michael Bruce-Lockhart for suggesting improvements to this article, to Athen O'Shea for careful editing, and to Ric Holt for introducing me to Hoare-style monitors many years ago.

References

[1] E. W. D. Dijkstra, "Hierarchical ordering of sequential processes," in *Operating System Techniques*. Academic Press, 1972.

[2] Per Brinch Hansen, *Operating System Principles*, Prentice-Hall, 1973.

[3] C. A. R. Hoare, "Monitors: an operating system structuring concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[4] John H. Howard, "Signaling in monitors," in *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, Los Alamitos, CA, USA, 1976, pp. 47–52, IEEE Computer Society Press.

[5] Ole-Johan Dahl, "Monitors revisited," in *A Classical Mind: Essays in Honour of C. A. R. Hoare*, A. W. Roscoe, Ed., chapter 6, pp. 93–103. Prentice Hall International, Hertfordshire, UK, 1994.

[6] Butler W. Lampson and David D. Redell, "Experience with processes and monitors in Mesa," *Commun. ACM*, vol. 23, no. 2, pp. 105–117, 1980.