Temporal Patterns

The patterns in this chapter involve coordinating concurrent operations. These patterns primarily address two different types of problems.

- Shared resources
  When concurrent operations access the same data or other type of resource, there may be the possibility that the operations may interfere with each other if they access the resource at the same time. To ensure that such operations execute correctly, the operations must be constrained to access their shared resource one at a time. However, if the operations are overly constrained, then they may deadlock and not be able to finish executing.

  Deadlock is a situation where one operation waits for another to do something before it proceeds. Because each operation is waiting for the other to do something, the wait forever and never do anything.

- Sequence of operations
  If operations are constrained to access a shared resource one at a time, then it may be necessary to ensure that the access the shared resource in a particular order. For example, an object cannot be removed from a data structure before it is added to the data structure.

The Single Threaded Execution pattern is the most important pattern in this chapter to know. Most shared resource issues can be resolved with just the Single Threaded Execution pattern. Situations where the sequence of operations matters less common.

Single Threaded Execution [Grand98]

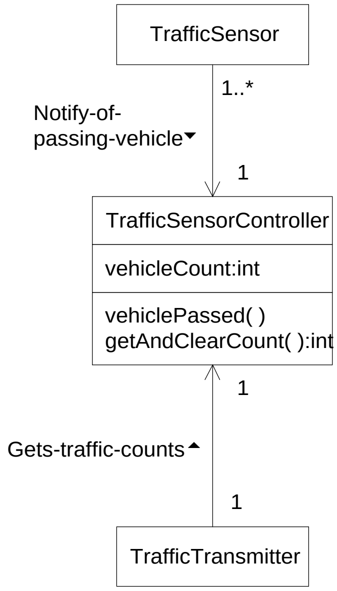
Synopsis

Some methods access data or other resources in a way that produces incorrect results if there are concurrent calls to a method and both calls access the data or other resource at the same time. The Single Threaded Execution pattern solves this problem by preventing concurrent calls to the method from resulting in concurrent executions of the method.

Context

Suppose you are writing software for a system that monitors the flow of traffic on a major highway. Sensors in the road monitor the number of passing cars per minute at strategic locations on the highway. The sensors send information to a central computer that controls electronic signs located near major interchanges. The signs display messages to drivers, advising them of traffic conditions so that they can select alternate routes.

At the places in the road where sensors measure the flow of cars, there is a sensor for each traffic lane. The sensor in each lane is wired to a controller that totals the number of cars that pass that place in the road each minute. The controller is attached to a transmitter that transmits each minute’s total to the central computer. Below is a class diagram that shows those relationships.



Traffic Sensor Classes

The above diagram contains the following classes:

TrafficSensor

Each instance of this class corresponds to a physical sensor device. Each time a vehicle passes a physical sensor device, the corresponding instance of the `TrafficSensor` class calls a `TrafficSensorController` object's `vehiclePassd` method

TrafficTransmitter

Instances of this class are responsible for transmitting the number of vehicles that pass a place on the road each minute. A `TrafficTransmitter` object gets the number of vehicles that have passed a place on the road by calling the `getAndClearCount` method of its corresponding `TrafficSensorController` object. The `TrafficSensorController` object's `getAndClearCount` method returns then number of vehicles that have passed the sensors since the previous call to the `getAndClearCount` method.

TrafficSensorController

Instances of the `TrafficSensor` class and the `TrafficTransmitter` class call the methods of the `TrafficSensorController` class to update, fetch and clear the number of vehicles that have passed a place on the road.

It is possible for two `TrafficSensor` objects to call a `TrafficSensorController` object's `vehiclePassed` method at the same time. If both calls execute at the same time, they produce an incorrect result. Each call to the `vehiclePassed` method is supposed to increase the vehicle count by one. However, if two calls to the `vehiclePassed` method execute at the same time, the vehicle count is incremented by one instead of two. Here is the sequence of events that would occur if both calls execute at the same time:

- Both calls fetch the same value of `vehicleCount` at the same time.
- Both calls add one to the same value.
- Both calls store the same value in `vehicleCount`.

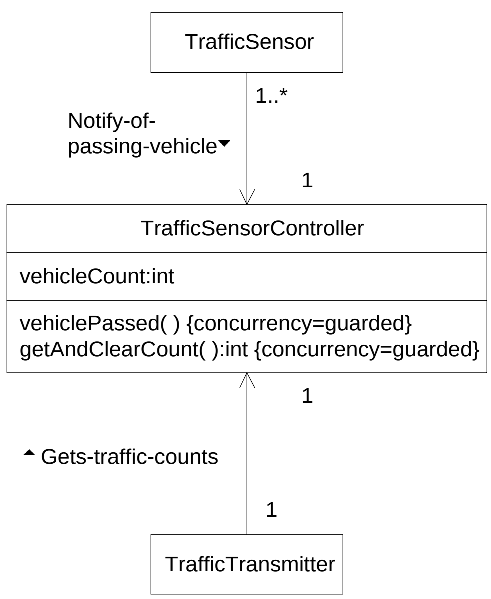
Clearly, allowing more than one call to the `vehiclePassed` method to execute at the same time will result in undercounting of passing vehicles. Though a slight undercount of vehicles may not be a serious problem for this application, there is a similar problem that is more serious.

A `TrafficTransmitter` object periodically calls a `TrafficSensorController` object's `getAndClearCount` method. The `getAndClearCount` method fetches the value of the `TrafficSensorController` object's `vehicleCount` variable and then sets it to zero. If a `TrafficSensorController` object's `vehiclePassed` method and `getAndClearCount` method are called at the same time, that creates a situation called a *race condition*.

A race condition is a situation whose outcome depends on the order in which concurrent operations finishes. If the `getAndClearCount` method finishes last, then it sets the value of the `vehicleCount` variable to zero, wiping out the result of the call to the `vehiclePassed` method. That is just another way for undercounts to happen. However, the problem is more serious if the `vehiclePassed` method finishes last.

If the `vehiclePassed` method finishes last, it replaces the zero set by the `getAndClearCount` method with a value one greater than the value it fetched. That means that the next call to the `getAndClearCount` method will return a value that includes vehicles counted before the previous call to the `getAndClearCount` method. An overcount like that could be large enough to convince the central computer that a traffic jam is starting and that it should display messages on the electronic signs suggesting that drivers follow alternate routes. An error like that could cause a traffic jam.

A simple way to avoid these problems is to require that no more than one thread at a time is executing a `TrafficSensorController` object's `vehiclePassed` method or `getAndClearCount` method at the same time. You can indicate that design decision by indicating that the concurrency of those methods is guarded: In a UML drawing, indicating that a method's concurrency is guarded is equivalent to declaring it synchronized in Java.



Synchronized Traffic Sensor Classes

Any number of threads may call the guarded methods of the same object at the same time. However, only one thread at a time is allowed to execute the object's guarded methods. While one thread is executing an object's guarded methods, other threads will wait until that thread is finished executing the object's guarded methods. That ensures single threaded execution of an object's guarded methods.

Forces

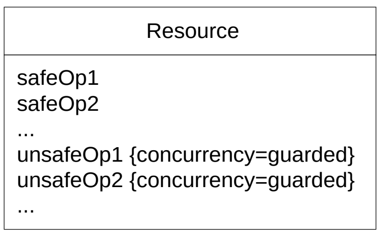
- A class implements methods that update or set instance or class variables.

- A class implements methods that manipulate external resources in a way that will only work correctly if the methods are executed by one thread at a time.

- The class' methods may be called concurrently by different threads.

Solution

The Single Threaded Execution pattern ensures that methods that perform operations that cannot be correctly performed concurrently are not performed concurrently. It accomplishes that by making methods that should not be executed concurrently guarded. The class diagram below shows the general case.



Single Threaded Execution Pattern

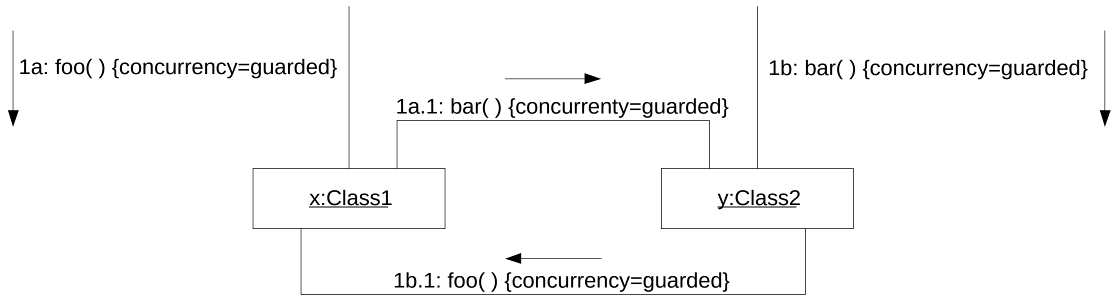
The class in the above has two kinds of methods. It has unguarded methods named safeOp1, safeOp2… that can be safely called concurrently by different threads. It has synchronized methods named unsafeOp1, unsafeOp2… that cannot be safely called concurrently by different threads. When different threads call the guarded methods of a `Resource` object at the same time, only one tread at a time is allowed to execute the method. The rest of the threads wait for that thread to finish.

Consequences

- If a class has methods that access variables or other resources in a way that is not thread safe, making all of its methods guarded that perform thread unsafe accesses to resources makes them thread safe.

- It is often the case that it takes longer to call a guarded method than an unguarded method. Making methods guarded that do not need to be can reduce performance.

- Making methods guarded can introduce the opportunity for threads to become *deadlocked*. Deadlock occurs when two threads each have the exclusive use of a resource and each thread is waiting for the other to release the other's resource before continuing. Since each thread is waiting for a resource that the other thread already has exclusive access to, both threads will wait forever without gaining access to the resource they are waiting for. Consider the example in the below collaboration diagram.



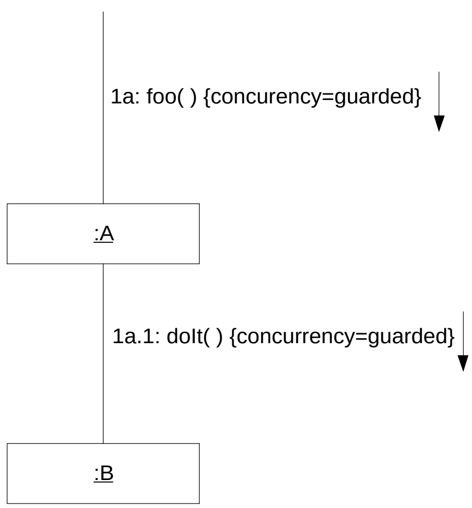
Deadlock

In the above collaboration diagram, at the same time, thread `1a` calls object `x`'s `foo` method and thread `1b` calls object `y`'s `bar` method. Thread `1a` then calls object `y`'s `bar` method and waits for thread`1b` to finish its call to that method. Thread `1b` calls object `x`'s `foo` method and waits for thread`1a` to finish its call to that method. At that point, the two threads are deadlocked. Each is waiting for the other to finish its call.

Deadlock can also involve more than two threads.

Implementation

Guarded methods are implemented in Java by declaring methods to be synchronized. It usually takes longer to call a synchronized method than an unsynchronized method. Consider the following collaboration diagram.



Synchronization Factoring

The above diagram shows that a synchronized method in class `A` calls class `B`'s `doIt` method. The `doIt` method is synchronized. If the `doIt` method is called only from synchronized methods of class `A`, then as an optimization, it is possible to make the `doIt` method an unsynchronized method. It will still be executed by only one thread at a time because it is only called by methods that are only executed by one thread at a time.

This optimization is called *synchronization factoring*. Synchronization factoring is an unsafe optimization in the sense that if the program is modified so that concurrent calls can be made to the `doIt` method, it will stop working correctly. For that reason, if you decide that this optimization is worth doing, you should put comments in the design diagrams and code to warn and remind people that the optimization has been performed.

JAVA API Usage

Many of the methods of the `java.util.Vector` class are synchronized to ensure single threaded access to the internal data structures of `Vector` objects.

Code Example

Below is some of the code that implements the traffic sensor design discussed under the “Context” heading. The first class shown is the `TrafficSensor` class. Instances of the `TrafficSensor` class are associated with a traffic sensor. A traffic sensor detects the passing of a vehicle over a place in a traffic lane. When the traffic sensor associated with an instance of the `TrafficSensor` class detects a passing vehicle, the instance's `detect` method is called. Its `detect` method is responsible for notifying other interested objects of the passing vehicle.

```
public class TrafficSensor implements Runnable {
    private TrafficObserver observer;

    /**
     * Constructor
     * @param observer The object to notify when this object’s associated
     *                 traffic sensor detects a passing vehicle.
     */
    public TrafficSensor(TrafficObserver observer) {
        this.observer = observer;
        new Thread(this).start();
    } // constructor(TrafficObserver)

    /**
     * Top level logic for this object’s thread.
     */
    public void run() {
        monitorSensor();
    } // run()

    // This method is responsible for calling this object’s detect when
    // its associated traffic sensor detects a passing vehicle.
    private native void monitorSensor() ;

     // This method is called by the monitorSensor method to report the
     // passing of a vehicle to this object’s observer.
    private void detect() {
        observer.vehiclePassed();
    } // detect()
...
    /**
     * Classes must implement this interface to be notified of passing
     * vehicles by a TrafficSensor object.
     */
    public interface TrafficObserver {
        /**
         * This is called when a TrafficSensor detects a passing vehicle.
         */
        public void vehiclePassed();
    } // interface TrafficObserver
} // class TrafficSensor
```

The next class shown is the `TrafficTransmitter` class. Instances of the `TrafficTransmitter` class are responsible for transmitting the number of vehicles that have passed a place in the road every minute.

```
public class TrafficTransmitter implements Runnable {
```

```
    private TrafficSensorController controller;
    private Thread myThread;

    /**
     * constructor.
     * @param controller The TrafficSensorController that this object
     *                   will get vehicle counts from.
     */
    public TrafficTransmitter(TrafficSensorController controller) {
        this.controller = controller;
        //...
        myThread = new Thread(this);
        myThread.start();
    } // constructor(TrafficSensorController)

    /**
     * transmit a vehicle count every minute
     */
    public void run() {
        while (true) {
            try {
                myThread.sleep(60*1000);
                transmit(controller.getAndClearCount());
            } catch (InterruptedException e) {
            } // try
        } // while
    } // run()

    // Transmit a vehicle count.
    private native void transmit(int count) ;
} // class TrafficTransmitter
```

The final class shown here is the `TrafficSensorController` class. Instances of the `TrafficSensorController` class maintain a running total of the number of vehicles that have passed the traffic sensors associated with the instance. Notice that its methods are implemented as synchronized methods.

```
public class TrafficSensorController
             implements TrafficSensor.TrafficObserver {
    private int vehicleCount = 0;
...
    /**
     * This method is called when a traffic sensor detects a passing
     * vehicle.  It increments the vehicle count by one.
     */
    public synchronized void vehiclePassed() {
        vehicleCount++;
    } // vehiclePassed()

    /**
     * Set the vehicle count to 0.
     * @return the previous vehicle count.
     */
    public synchronized int getAndClearCount() {
        int count = vehicleCount;
        vehicleCount = 0;
        return count;
    } // getAndClearCount()
} // class TrafficSensorController
```

Related Patterns

Most other temporal patterns use the Single Threaded Execution pattern.

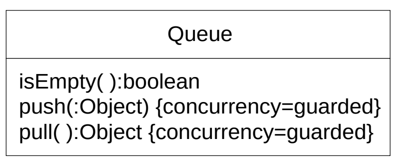
Guarded Suspension ***[Lea97]***

Synopsis

Suspend execution of a method call until a precondition is satisfied.

Context

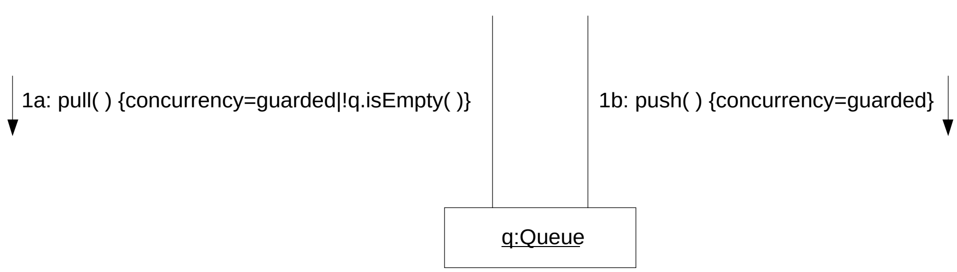
Suppose that you have to create a class that implements a queue data structure. A queue is a first in first out data structure. Objects are removed from a queue in the same order that they are added. The following class diagram shows a `Queue` class.



Queue Class

The class in the above diagram has two methods. The `push` method adds objects to a queue and the `pull` method removes objects from the queue. When the queue is empty, you want the `get` method to wait until a call to the add an object to the queue for it to return. The methods are synchronized to allow concurrent access to a `Queue` object. Simply making both methods synchronized creates a problem when there is a call to a `Queue` object's `pull` method and the queue is empty. The `pull` method waits for a call to the `push` method to provide it with an object to return. However, because they are both synchronized, calls to the `push` method cannot execute until the `pull` method returns and the `pull` method will never return until a call to the `push` method executes.

A solution to the problem is to add a precondition to the `pull` method so that it does not execute when the queue is empty. Consider the following collaboration diagram.



Queue Collaboration

The above collaboration diagram shows concurrent calls to a `Queue` object's `push` and `pull` methods. If the `pull` method is called when the `Queue` object's `isEmpty` method returns true, then the thread waits until `isEmpty` returns false before executing the `pull` method. Because it does not actually execute the `pull` method while the queue is empty, there is no problem with a call to the `push` method being able to add objects to an empty queue.

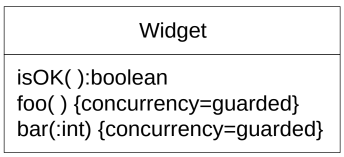
Forces

- A class' methods must be synchronized to allow safe concurrent calls to them.

- An object may be in a state that will make it impossible for one of its synchronized methods to execute to completion. In order for the object to leave that state, a call to one of the object's other synchronized methods must execute. If a call to the first method is allowed to proceed it while the object is in that state, it will never complete. Calls to the method that allows it to complete will have to wait until it does complete, which will never happen.

Solution

Consider the following class diagram.

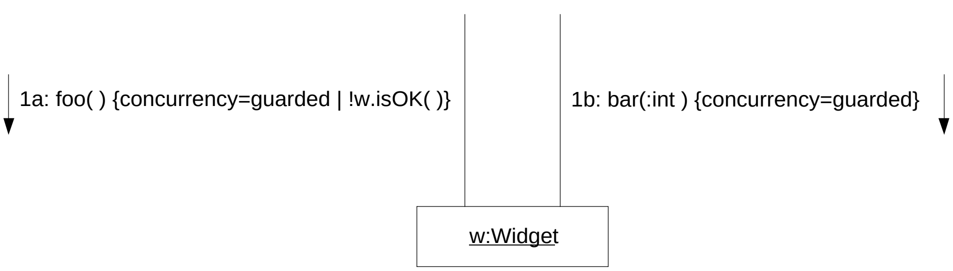


Unguarded Suspension Class

The above diagram shows a class named `Widget` that has two synchronized methods named `foo` and `bar`. There is an exceptional state that `Widget` objects can enter. When a `Widget` object enters that exceptional state, its `isOK` method returns false; otherwise, it returns true. When a widget object enters the exceptional state, a call to its `bar` method may take it out of that state. There is no way to take it out of the exceptional state other than a call to `bar`. Taking a `Widget` object out of its exceptional state is a side effect of the `bar` method's main purpose, so it is not acceptable to call the `bar` method just to take a widget object out of its exceptional state.

A call to a `Widget` object's `foo` method cannot complete if the `Widget` object is in its exceptional state. If that happens, because the `foo` and `bar` methods are synchronized, subsequent calls to the `Widget` object's `foo` and `bar` methods will not execute until the call to `foo` returns. The call to `foo` will not return until a call to bar takes the `Widget` object out of its exceptional state.

The purpose of the Guarded Suspension pattern is to avoid the deadlock situation that can occur when a thread is about to execute an object's synchronized method and the state of the object prevents the method from completing. If a method call occurs when an object is in a state that prevents the method from executing to completion, the Guarded Suspension pattern suspends the thread until the object is in a state that allows the method to complete. That is illustrated in the collaboration diagram below.



Guarded Suspension Collaboration

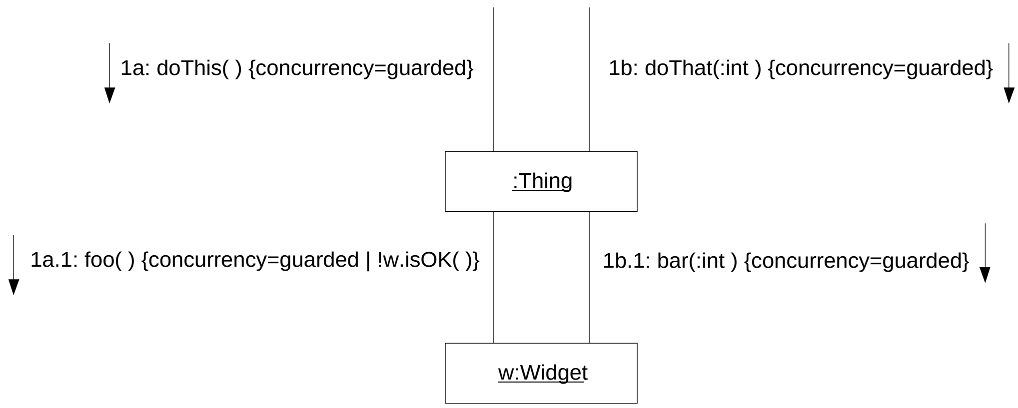
Notice that the above diagram indicates a precondition that must be satisfied before a call to a `Widget` object executes. If a thread tries to call a `Widget` object's `foo` method when the `Widget` object's `isOK` method returns false, the thread will be forced to wait until `isOK` returns true before it is able to execute the `foo` method. While that thread is waiting for `isOK` to return true, other threads are free to call the `bar` method.

Consequences

Using the Guarded Suspension pattern allows a thread to cope with an object that is in the wrong state to perform an operation by waiting until the object is in a state that allows it to perform the operation.

Because the Guarded Suspension pattern requires synchronized methods, it is possible for multiple threads to be waiting to execute a call to the same method of the same object. The Guarded Suspension pattern specifically does not deal with selecting which of the waiting threads will be allowed to proceed when the object is in a state that will allow the method to be executed. You can use the Scheduler pattern to accomplish that.

A situation that can make it difficult or impossible to use the Guarded Synchronization pattern is nested synchronization. Consider the following collaboration diagram.



Guarded Suspension under Nested Synchronization

In the diagram above, access to the `Widget` object is through the synchronized methods of a `Thing` object. That means that when thread `1a` calls the `Widget` object's `foo` method when the state of the `Widget` object causes its `isOK` method to return false, the thread will wait forever for the `Widget` object's `isOK` method to return true. The reason for that is that the methods of the `Thing` object are synchronized without any preconditions. That gives us the same problem that the Guarded Suspension pattern was intended to solve.

Implementation

The Guarded Suspension pattern is implemented using the `wait` and `notify` methods like this:

```
class Widget {
    synchronized void foo(){
        while (!isOK()) {
            wait();
        }
        ...
    }

    synchronized void bar(x int) {
        ...
        notify();
    }
}
```

The way that it works is that a method such as `foo` that must satisfy preconditions before it begins executing actually begins executing and then tests its preconditions. The very first thing that such a method does is to test its preconditions in a while loop. While the preconditions are false, it calls the `wait` method.

Every class inherits the `wait` method from the `Object` class. When a thread calls an object's `wait` method, the `wait` method causes the thread to release the synchronization lock it holds on the object. The method then waits until it is notified that it may return. Then, as soon as the thread is able to recapture the lock, the `wait` method returns.

When the `wait` method returns, control returns to the top of the while loop which tests the preconditions again. The reason for testing the preconditions in a loop is that between the time that the thread first tries to recapture the synchronization lock and the time that it captures it, another thread may have made the preconditions false.

The call to the `wait` method is notified that it should return when another method, such as `bar`, calls the object's `notify` method. Such methods call the `notify` method after they have changed the state of the object in a way that may satisfy a method's preconditions. The `notify` method is another method that all classes inherit from the `Object` class. What the `notify` method does is to notify another thread that is waiting for the `wait` method to return that it should return.

If more than one thread is waiting, the `notify` method chooses one arbitrarily. Arbitrary selection works well in most situations. It does not work well for objects that have methods with different preconditions. Consider a situation in which multiple method calls are waiting to have their different preconditions satisfied. Arbitrary selection can result in a situation where the preconditions of one method call are satisfied, but the thread that gets notified has is trying to execute a method with different preconditions that are not satisfied. In a situation like that it is possible for a method call to never complete because the method is never notified when its preconditions are satisfied.

For classes where arbitrary selection is not a good way to decide which thread to notify, there is an alternative. Their methods can call the `notifyAll` method. Rather than choosing one thread to notify, the `notifyAll` method notifies all waiting threads. That avoids the problem of not notifying the right thread. However, it may result in wasted machine cycles as a result of waking up threads waiting to execute method calls whose preconditions are not satisifed..

Code Example

Below is code that implements the `Queue` class design discussed under the "Context" heading.

```
public class Queue {
    private Vector data = new Vector();

    /**
     * Put an object on the end of the queue
     * @param obj the object to put at end of queue
     */
    synchronized public void put(Object obj) {
        data.addElement(obj);
        notify();
    } // put(Object)

    /**
     * Get an object from the front of the queue
     * If queue is empty, waits until it is not empty.
     */
    synchronized public Object get() {
```

```
        while (data.size() == 0){
            try {
                wait();
            } catch (InterruptedException e) {
            } // try
        } // while
        Object obj = data.elementAt(0);
        data.removeElementAt(0);
        return obj;
    } // get()
} // class Queue
```

Notice that in the above listing, the call to the `wait` method is wrapped in a `try` statement that catches the `InterruptedException` that calls to the `wait` method may throw. Simply ignoring the `InterruptedException` that the `wait` method is declared to throw is the simplest thing to do for programs that do not expect the `wait` method to actually throw an `InterruptedException`. See the discussion of the Two-Phase Termination pattern for an explanation of when the `wait` method throws an `InterruptedException` and what you should do about it.

Related Patterns

Balking
: The Balking pattern provides a different strategy for handling method calls to objects that are not in an appropriate state to execute the method call.

Two-Phase Termination
: Because the implementation of the Two-Phase Termination pattern usually involves the throwing and handling of `InterruptedException`, its implementation usually interacts with the Guarded Suspension pattern.

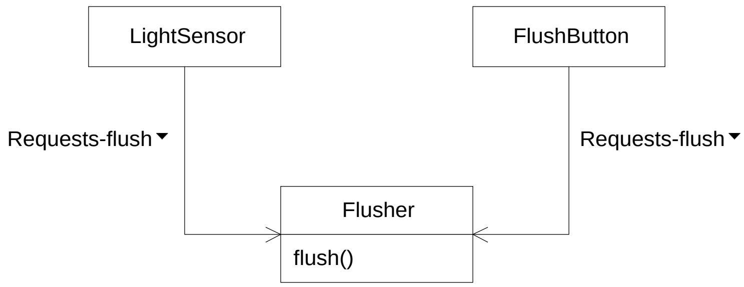
***Balking* [Lea97]**

Synopsis

If one of an object's methods is called when the object is not in an appropriate state to execute that method, have the method return without doing anything.

Context

Suppose that you are writing a program to control an electronic toilet flusher. Such devices are intended for use in public bathrooms. They have a light sensor mounted on the front of the flusher. When the light sensor detects an increase in the light level, it assumes that a person has left the toilet and triggers a flush. Electronic toilet flushers also have a button on them that can be used to manually trigger a flush. Below is a class diagram showing classes to model that behavior.



Flusher Classes

As shown in the diagram above, when a `LightSensor` object or a `FlushButton` object decide that there should be a flush, they request the `Flusher` object to start a flush. They do that by calling the `Flusher` object's `flush` method. The `flush` method starts a flush and then returns once the flush is started. That arrangement raises some concurrency issues that need to be resolved.

You will need to decide what happens when the `flush` method is called while there is already a flush in progress. You will also need to decide what happens when both the `LightSensor` object and the `FlushButton` object call the `Flusher` object's `flush` method at the same time.

The three most obvious choices for how to handle a call to the `flush` method while there is a flush in progress are:

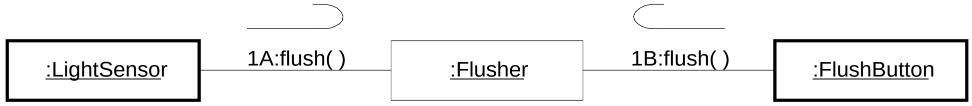
- Start a new flush immediately.
  Starting a new flush while a flush is already in progress has the same effect as making the flush in progress last longer than a normal flush. The optimal length of a normal flush has been determined through experience. A longer flush would be a waste of water, so this is not a good option.

- Wait until the current flush finishes and immediately start another flush.
  This option effectively doubles the length of a flush, so it is a bigger waste of water that the first option.

- Do nothing.
  This option wastes no water, so it is the best choice.

When there are two concurrent calls to the flush method, allowing one to execute and ignoring the other is also a good strategy.

Suppose that a call is made to an object's method when the object is not in a state to properly execute the method. If the method handles the situation by returning without performing its normal function, we say that the method balked. UML does not have a standard way of indicating a method call with balking behavior. The technique used in this book to represent a method call that exhibits balking behavior in a collaboration diagram is an arrow that curves back on itself, as shown below.



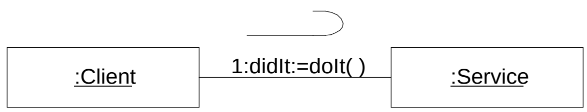
Flusher Collaboration

Forces

- An object may be in a state in which it is inappropriate to execute a method call.
- Postponing execution of a method call until the object is in an appropriate state is an inappropriate policy for the problem at hand.
- Calls made to an object’s method when the object is not in an appropriate state to execute the method may be safely ignored.

Solution

The following collaboration diagram shows objects collaborating in the Balking pattern.



Balking Collaboration

In the above diagram, a `Client` object calls the `doIt` method of a `Service` object. The bent back arrow indicates that the call may balk. If the `Service` object’s `doIt` method is called when the `Service` object is in a state that is inappropriate for executing a call to its `doIt` method, then the method returns without have performed its usual functions.

The `doIt` method returns a result, indicated in the diagram as `didIt`, that indicates if the method performed its normal functions or balked.

Consequences

- Method calls are not executed if they are made when their object is in an inappropriate state.
- Calling a method that can balk means that the method may not perform the functions that its caller expects that it will, but do nothing instead.

Implementation

If a method can balk then, generally, the first thing that it does is to check the state of the object it belongs to, to determine if it should balk. It may be possible for the object’s state to change to an inappropriate state for a balking method to run while that method is running. If that is possible, then an application of the Single Threaded Execution pattern can be used to prevent that inconsistency.

Instead of telling its callers if it balked by passing a return value, it is also reasonable for a method to notify its callers that it balked by throwing an exception. If a method’s callers do not need to be interested in whether or not it balked, the method does not need to return that information.

Code Example

Below is code for the `Flusher` class discussed under the “Context” heading.

```
public class Flusher {
    private boolean flushInProgress = false;

    /**
```

```
     * This method is called to start a flush.
     */
    public void flush() {
        synchronized (this) {
            if (flushInProgress)
              return;
            flushInProgress = true;
        }
        // code to start flush goes here.
    }

    /**
     * This method is called to notify this object that a flush has completed.
     */
    void flushCompleted() {
        flushInProgress = false;
    } // flushCompleted()
} // class Flusher
```

Notice the use of the synchronized statement in the `flush` method. It is there to ensure that if two calls to the `flush` method occur at the same time, one of the calls will proceed normally and the other will balk.

Also, notice that the `flushCompleted` method is not synchronized. That is because there is never a time when setting the `flushInProgress` variable to false causes a problem.

Related Patterns

Guarded Suspension

The Guarded Suspension pattern provides an alternate way to handle method calls to objects that are not in an appropriate state to execute the method call.

Single Threaded Execution

The Balking pattern is often combined with the Single Threaded Execution pattern to coordinate changes to an object's state.

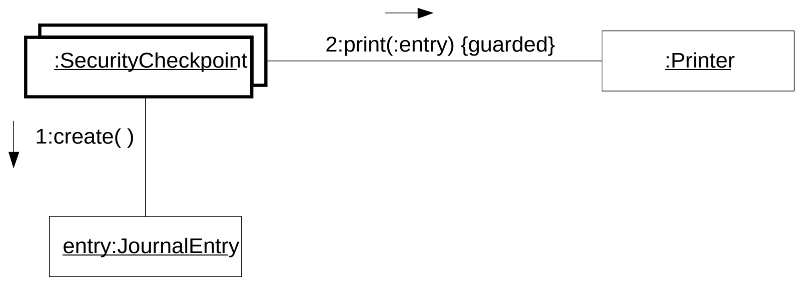
Scheduler [Lea97]

Synopsis

Control the order in which threads are scheduled to execute single threaded code using an object that explicitly sequences waiting threads.

Context

Suppose that you are designing software to manage a building's physical security. The security system will support security checkpoints that require a person to pass their identification badge through a scanner in order to pass through the checkpoint. When someone passes their identification badge through a checkpoint scanner, the checkpoint either allows the person to pass through or rejects the badge. Whenever someone passes through a security checkpoint or a badge is rejected, an entry is to be printed on a hard copy log in a central security office. Here is a diagram showing the basic collaboration.

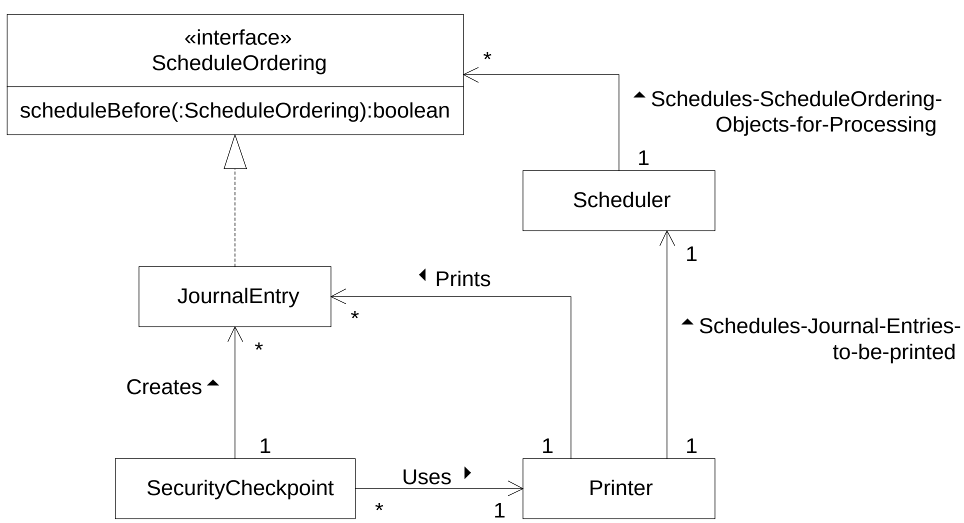


Security Journal Collaboration

The above interaction diagram shows `SecurityCheckpoint` objects creating `JournalEntry` objects and passing them to a `Printer` object's `print` method. Simple though it is, there is a problem with that organization. The problem occurs when people go through three or more checkpoints at the same time. While the printer is printing the first of the log entries, the other print calls wait. After the first log entry is printed, there is no guarantee which log entry will be printed next. That means that the log entries may not be printed in the same order that the security checkpoints sent them to the printer.

To ensure that journal entries are printed in the same order that they happen, you could simple put each journal entry in a queue and then print the journal entries in the same order that the arrive in a queue. Though that still leaves open the possibility of three or more journal entries arriving at the same time, the likelihood is greatly reduced. It may take as long as a second to print a journal entry. For the problem to occur, the other two journal entries must both arrive within that time period. Queuing a journal entry may only take about a microsecond. That reduces likelihood of journal entries printing out of sequence by a factor of 1,000,000.

You could make the queuing of journal entries to be printed the responsibility of the `Printer` class. However, the queuing of method calls to be executed sequentially is a capability that has a lot of potential reuse if it is implemented as a separate class. The interaction diagram below shows how a printer object could collaborate with another object to queue the execution of calls to its `print` method.



Security Journal with Scheduler

The above interaction, a `SecurityCheckpoint` object calls the `printer` object's `print` method. The `print` method begins by calling the `Scheduler` object's `enter` method. The `enter` method does not return until the `Scheduler` object decides that it should. When the `print` method is finished, it calls the `Scheduler` object's `done` method. Between the time that the `enter` method returns and the `done` method is called, the `Scheduler` object assumes that the resource it is managing is busy. No call to its `enter` method will return while the

Scheduler object believes that the resource it is managing is busy. That ensures that only one thread at a time executes the portion of the print method after its call to the Scheduler object's enter method until it calls the Scheduler object's done method.

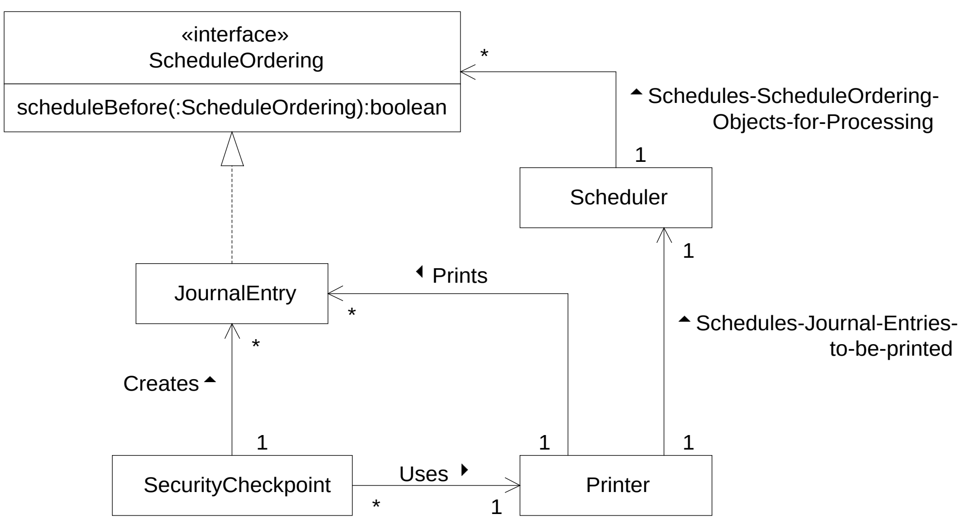
The actual policy that the Scheduler object uses to decide when a call to the enter method returns is encapsulated in the Scheduler object. That allows the policy to change without affecting other objects. In this example, the policy that you would want when more than one call to the enter method is waiting to return is

- If the Scheduler object is not waiting for a call to its done method, then a call to its enter method will return immediately. The Scheduler object then waits for a call to its done method.

- If the Scheduler object is waiting for a call to its done method, then a call to its enter method will wait to return until a call to the Scheduler object's done method. When the Scheduler object's done method is called, if there are any calls to its enter method waiting to return then one of those enter method calls is choose to return.

- The enter method call chosen to return when the done method is called is the one that was passed a JournalEntry object with the earliest timestamp. If more than one JournalEntry object has the same earliest timestamp, then one of them is chosen arbitrarily.

In order for the Scheduler class to be able to compare the timestamps of JournalEntry objects and still be reusable, the Scheduler class must not refer directly to the JournalEntry class. However, it can refer to the JournalEntry class through an interface and still remain reusable. That organization is shown in the class diagram below.



Journal Entry Scheduling Classes

The Scheduler class does not know about the JournalEntry class. It merely knows that it schedules processing for objects that implement the ScheduleOrdering interface. That interface declares the scheduleBefore method that the Scheduler class calls to determine which of two ScheduleOrdering objects it should schedule first. Though the Scheduler class encapsulates a policy governing when processing will be allowed for a ScheduleOrdering object, it delegates the decision of what order they will be allowed to process in to the ScheduleOrdering object.

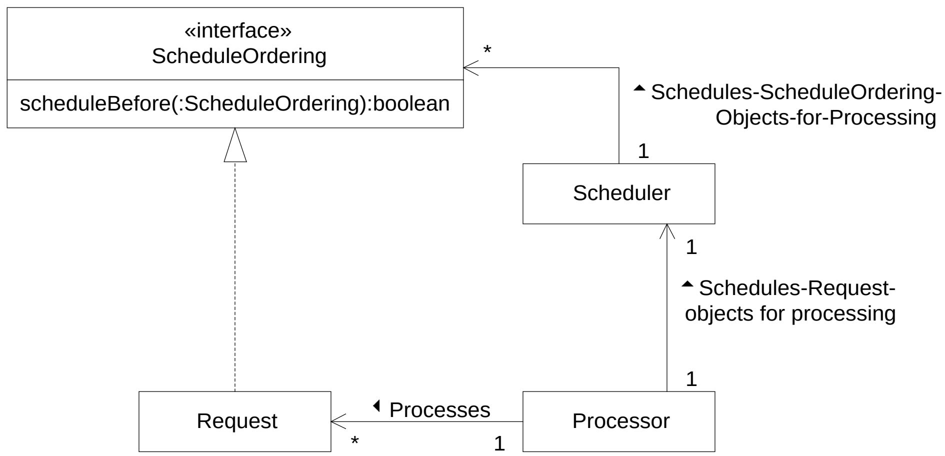
Forces

- Multiple threads may need to access a resource at the same time and only one thread at a time may access the resource.

- The program's requirements imply constraints on the order in which threads should access the resource.

Solution

The `Scheduler` pattern uses an object to explicitly schedule concurrent requests by threads for non-concurrent processing. The class diagram below shows the roles that classes play in the Scheduler pattern.



Scheduler Classes

Below are descriptions of the role the classes play in the Scheduler pattern.

Request
: Classes in this role must implement the interface in the `ScheduleOrdering` role. `Request` objects encapsulate a request for a `Processor` object to do something.

Processor
: Instances of classes in this role perform a computation described by a `Request` object. They may be presented with more that one `Request` object to process at a time, but can only process one at a time. A `Processor` object delegates to a `Scheduler` object the responsibility for scheduling `Request` object for processing, one at a time.

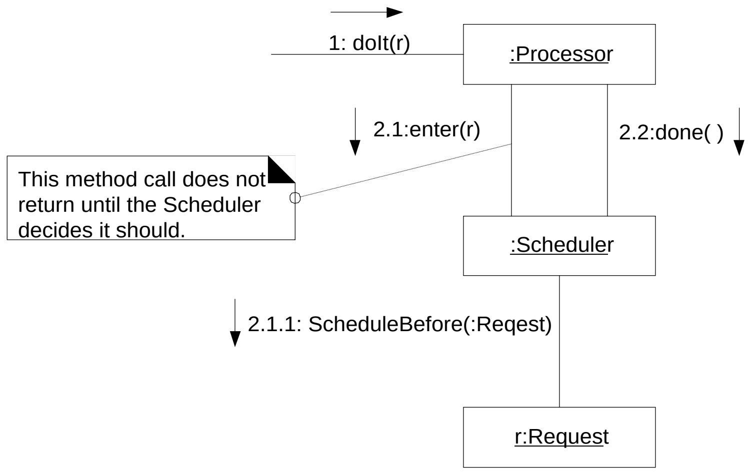
Scheduler
: Instances of classes in this role schedule `Request` objects for processing by a `Processor` object. To promote reusability, a Scheduler class does not have any knowledge of the `Request` class that it schedules. Instead, it accesses `Request` objects through the `ScheduleOrdering` interface that they implement.

ScheduleOrdering
: `Request` objects implement an interface that is in this role. Interfaces in this role serve two purposes.

- By referring to a `ScheduleOrdering` interface, `Processor` classes avoid a dependency on a `Request` class.

- By calling methods defined by the `ScheduleOrdering` interface, `Processor` classes are able to delegate the decision of which `Request` object to schedule for processing next, which increases the reusability of `Processor` classes. The above class diagram indicates one such method named `scheduleBefore`.

The interaction between a `Processor` object and a `Scheduler` object occurs in two stages, as is shown by the interaction diagram below.



Scheduler Interaction

The above interaction begins with a call to a `Processor` object's `doIt` method. The very first thing the `doIt` method does is to call the `enter` method of the `Scheduler` object associated with the `Processor` object. If there is no other thread currently executing the rest of the `doIt` method, then the `enter` method returns immediately. After the `enter` method returns, the `Scheduler` object knows that the resource that it is managing is busy. While its resource is busy, any calls to the `Scheduler` object's `enter` method will not return until the resource is not busy and the `Scheduler` object decides that it is that call's turn to return.

After its enter method returns, a `Scheduler` object considers the resource it manages to be busy until the `Scheduler` object's `done` method is called. When one thread makes a valid call to a `Scheduler` object's `done` method, if any threads are waiting to return from the `Scheduler` object's `enter` method then one of them returns.

If a call to a `Scheduler` object's `enter` method must wait before it returns and there are other calls waiting to return from the `enter` method then the `Scheduler` object must decide with call will return next. It decides by consulting the `Request` objects that were passed into those calls to decide which call will return next. It does that indirectly by calling methods declared for that purpose by the `ScheduleOrdering` interface and implemented by the `Request` object.

Consequences

- The Scheduler pattern provides a way to explicitly control when threads may execute a piece of code.
- The scheduling policy is encapsulated in its own class and is reusable.
- Using the Scheduler pattern adds significant overhead beyond what is required to make a simple call to a synchronized method.

Implementation

In some applications of the Scheduler pattern, the `Scheduler` class implements a scheduling policy that does not require it to consult `Request` objects to determine the order in which calls to its `enter` method will return. An example of such a policy is to allow calls to the `enter` method to return in the order in which they were called. In such cases, there is no need to pass `Request` objects into the `enter` method or to have a `ScheduleOrdering` interface. Another example of such a policy, is to not care about the order in which requests are scheduled but require at least five minutes between the end of one task and the beginning of another.

Code Example

Below is some of the code that implements the print scheduling design discussed under the "Context" heading. The first listing is of the `Printer` class that manages the printing of security checkpoint journal entries.

```
class Printer {
    private Scheduler scheduler = new Scheduler();

    public void print(JournalEntry j) {
        try {
            scheduler.enter(j);
            ...
        } catch (InterruptedException e) {
        } //try
        scheduler.done();
    } // print(JournalEntry)
} // class Printer
```

Each `Printer` object uses a `Scheduler` object to schedule concurrent calls to its `print` method so that they print sequentially in the order of their timestamps. It begins by calling the `Scheduler` object's `enter` method, passing it the `JournalEntry` object to be printed. The call does not return until the `Scheduler` object decides that it is the `JournalEntry` object's turn to print.

The `print` method ends by calling the `Scheduler` object's `done` method. A call to the `done` method tells the `Scheduler` object that the `JournalEntry` object has been printed and another `JournalEntry` object can have its turn to be printed.

Below is the source for the `Scheduler` class.

```
public class Scheduler {
    private Thread runningThread;
```

The `runningThread` variable is `null` when the resource that a `Scheduler` object manages is not busy. It contains a reference to the thread using the resource when the resource is busy.

```
    private ArrayList waitingRequests = new ArrayList();
    private ArrayList waitingThreads = new ArrayList();
```

An invariant for this class is that a request and its corresponding thread are only in `waitingRequests` and `waitingThreads` while its call to the `enter` method is waiting to return.

The `enter` method is called before a thread starts using a managed resource. The `enter` method does not return until the managed resource is not busy and this `Scheduler` object decides it is the method call's turn to return.

```
    public void enter(ScheduleOrdering s) throws InterruptedException {
        Thread thisThread = Thread.currentThread();

        // For the case when the managed resource is not busy,
        // synchronize on this object to ensure that two concurrent
        // calls to enter do not both return immediately.
        synchronized (this) {
            if (runningThread == null) {
                runningThread = thisThread;
                return;
```

```
            } // if
            waitingThreads.add(thisThread);
            waitingRequests.add(s);
        } // synchronized (this)
        synchronized (thisThread) {
            while (thisThread != runningThread) {
                thisThread.wait();
            } // while
        } // synchronized (thisThread)
        synchronized (this) {
            int i = waitingThreads.indexOf(thisThread);
            waitingThreads.remove(i);
            waitingRequests.remove(i);
        } // synchronized (this)
    } // enter(ScheduleOrdering)
```

A call to the `done` method indicates that the current thread is finished with the managed resource.

```
    synchronized public void done() {
        if (runningThread != Thread.currentThread())
          throw new IllegalStateException("Wrong Thread");
        int waitCount = waitingThreads.size();
        if (waitCount <= 0)
          runningThread = null;
        else if (waitCount == 1) {
            runningThread = (Thread)waitingThreads.get(0);
            waitingThreads.remove(0);
        } else {
            int next = waitCount - 1;
            ScheduleOrdering nextRequest;
            nextRequest = (ScheduleOrdering)waitingRequests.get(next);
            for (int i = waitCount-2; i>=0; i--) {
                ScheduleOrdering r;
                r = (ScheduleOrdering)waitingRequests.get(i);
                if (r.scheduleBefore(nextRequest)) {
                    next = i;
                    nextRequest = (ScheduleOrdering)waitingRequests.get(next);
                } // if
            } // for
            runningThread = (Thread)waitingThreads.get(next);
            synchronized (runningThread) {
                runningThread.notifyAll();
            } // synchronized (runningThread)
        } // if waitCount
    } // done()
} // class Scheduler
```

The done method uses the `notifyAll` method to wake up a thread, rather than the `notify` method, because it has no guarantee that there will not be another thread waiting to regain ownership of the lock on the `runningThread` object. If it used the `notify` method, and there were additional threads waiting to regain ownership of the `runningThread` object's lock, then the `notify` method could fail to wake up the right thread.

Related Patterns

Read/Write Lock

Implementations of the Read/Write Lock pattern usually use the Scheduler pattern to ensure fairness in scheduling.

Read/Write Lock *[Lea97]*

Synopsis

Allow concurrent read access to an object but require exclusive access for write operations.

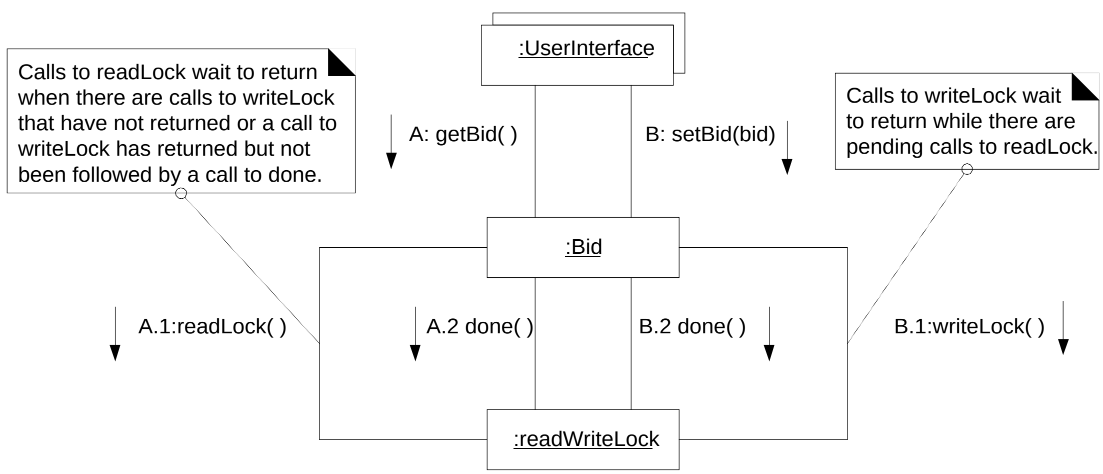
Context

Suppose that you are developing software for conducting online auctions. The way that these auctions will work is that an item will be put up for auction. People will access the online auction to see what the current bid for an item is. People can then decide to make a bid for an item that is greater than the current bid. At a predetermined time, the auction closes and the highest bidder at that time get the item at the final bid price.

You expect that there will be many more requests to read the current bid for an item than to update it. You could use the Single Threaded Execution pattern to coordinate access to bids. Though that will ensure correct results, it can unnecessarily limit responsiveness. When multiple users want to read the current bid on an item at the same time, Single Threaded Execution means that only one user at a time is allowed to read the current bid. Users who just want to read the current bid are forced to wait for other users who just want to read the current bid.

There is no reason to prevent multiple users from reading the current bid at the same time. Single threaded execution is only required for updates to the current bid. Updates to the current bid must be processed one at a time to ensure that updates that would not increase the value of the current bid are ignored.

The Read/Write Lock pattern avoids unnecessary waiting to read data by allowing concurrent reads of data but only allowing single threaded access to data when it is being updated. Consider the following interaction diagram.



Bid Collaboration

The above collaboration diagram shows multiple user interface objects calling a bid object's `getBid` and `setBid` methods. The `getBid` method waits until there are no calls to `setBid` waiting to complete before it returns the current bid. The `setBid` method waits for any executing calls to `getBid` or `setBid` to complete before it updates the current bid. The `readWriteLock` object encapsulates the logic that coordinates the execution of the `getBid` and `setBid` methods to allow it to be reused.

All calls to a `readWriteLock` object's `readLock` method return immediately, unless there are any calls to its `writeLock` method executing or waiting to execute. If any calls to a `readWriteLock` object's `writeLock` method are executing or waiting to execute, then calls to its `readLock` method wait to return. They wait until all the `writeLock` calls have completed and there have been corresponding calls to its `done` method.

Calls to a `readWriteLock` object's `writeLock` method return immediately, unless one or more of the following are true:

- A previous call to `writeLock` is waiting to execute.
- A previous call to `writeLock` has finished executing, but there has been no corresponding call to the `readWriteLock` object's `done` method.
- There are any executing calls to the `readWriteLock` object's `readLock` method.

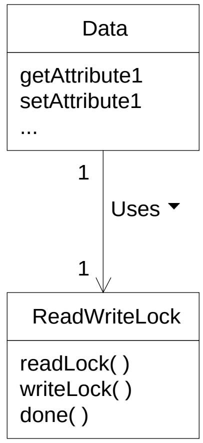
If a call to a `readWriteLock` object is made when any of the above conditions are true, it will not return until all of the above conditions are false.

Forces

- There is a need for read and write access to an object's state information.
- Any number of reads operations may be performed on the object's state information concurrently. However, read operations are only guaranteed to return the correct value if there are no write operations executing at the same time as a read operation.
- Write operations on the object's state information need to be performed one at a time, to ensure their correctness.
- There will be more read operations than write operations on the object's state information. Many of the reads will be initiated concurrently.
- Allowing concurrently initiated read operations to run concurrently will improve responsiveness and throughput.
- The logic for coordinating read and write operations should be reusable.

Solution

The Read/Write Lock pattern organizes a class so that concurrent calls to methods that fetch and store its instance information are coordinated by an instance of another class. The following class diagram shows the roles that classes play in the Read/Write Lock pattern.



Read-Write Lock Classes

A class in the `Data` role has methods to get and set its instance information. Any number of threads are allowed to concurrently get a `Data` object's instance information, so long as no thread is setting its instance information at the same time. On the other hand, its set operations must occur one at a time, while there are no get operations being executed. `Data` objects must coordinate their set and get operations so that they obey those restrictions.

The abstraction that `Data` objects use to coordinate get operations is a read lock. A `Data` object's get methods do not fetch any information until they get a read lock. Associated with each `Data` object is a `ReadWriteLock` object. Before one of its get methods gets anything, it calls the associated `ReadWriteLock` object's `readLock` method, which issues a read lock to the current thread. While the thread has a read lock, the get method can be sure that it is safe for it to get data from the object. That is because while there are any outstanding read locks, the `ReadWriteLock` object will not issue any write locks. If there are any outstanding write locks when the `ReadWriteLock` object's `readLock` method is called, it does not return until all the outstanding write locks have been relinquished by calls to the `ReadWriteLock` object's `done` method. Otherwise, calls to the `ReadWriteLock` object's `readLock` method return immediately.

When a `Data` object's get method is finished getting data from the object, it calls the `ReadWriteLock` object's `done` method. A call to that method causes the current thread to relinquish its read lock.

Similarly, `Data` objects use a write lock abstraction to coordinate set operations. A `Data` object's set methods do not store any information until they get a write lock. Before one of a `Data` object's set methods store any information, it calls the associated `ReadWriteLock` object's `writeLock` method, which issues a write lock to the current thread. While the thread has a write lock, the set method can be sure that it is safe for it to store data in the object. That is because the `ReadWriteLock` object only issues write locks when there are no outstanding read locks and no outstanding write locks. If there are any outstanding locks when the `ReadWriteLock` object's `writeLock` method is called, it does not return until all of the outstanding locks have been relinquished by calls to the `ReadWriteLock` object's `done` method.

The preceding constraints that govern when read and write locks are issued do not address the order in which read and write locks are issued. The order in which read locks are issued does not matter, so long as get operations can be performed concurrently. Since write operations are performed one at a time, the order in which write locks are issued should be the order in which the write locks are requested.

The one remaining ambiguity occurs when there are calls to both of a `ReadWriteLock` object's `readLock` and `writeLock` methods waiting to return and there are no outstanding locks. If get operations are intended to return the most current information, then that situation should result in the `writeLock` method returning first.

Consequences

- The Read/Write Lock pattern coordinates concurrent calls to an object's get and set methods so that calls to the object's set methods do not interfere with each other or calls to the object's get methods.

- If there are many concurrent calls to an object's get methods, using the Read/Write Lock pattern to coordinate the calls can result in better responsiveness and throughput than using the Single Threaded Execution pattern for that purpose. That is because the Read/Write Lock pattern allows concurrent calls to the object's get methods to execute concurrently.

- If there are relative few concurrent calls to an object's get methods, using the Read/Write Lock pattern will result in lower throughput than using the Single Threaded Execution pattern. That is because the Read/Write Lock pattern spends more time managing individual calls. When there are concurrent get calls for it to manage, that results in a net improvement.

Implementation

Since read locks and write locks do not contain any information, there is no need to represent them as explicit objects. It is sufficient to just count them.

Code Example

Below is code the implements the design discussed under the "Context" heading. The first listing is the `Bid` class, which is rather straightforward.

```
public class Bid {
    private int bid = 0;
    private ReadWriteLock lockManager = new ReadWriteLock();
...
    public int getBid() throws InterruptedException{
        lockManager.readLock();
        int bid = this.bid;
        lockManager.done();
        return bid;
    } // getBid()

    public void setBid(int bid) throws InterruptedException {
        lockManager.writeLock();
        if (bid > this.bid) {
            this.bid = bid;
        } // if
        lockManager.done();
    } // setBid(int)
} // class Bid
```

As you can see, the methods of the `Bid` class simply use a `ReadWriteLock` object to coordinate concurrent calls. They begin by calling the appropriate lock method before getting or setting any values. When they are finished, they call the `ReadWriteLock` object's `done` method to release the lock.

The `ReadWriteLock` class is more complex. As you read through its listing, you will notice that there are two main things it focuses on.

- It carefully tracks state information in a way that will be consistent for all threads.

* It ensures that all preconditions are met before its lock methods return.

Any other class that is responsible for enforcing a scheduling policy will have these implementation concerns.

```
public class ReadWriteLock {
    private int waitingForReadLock = 0;
    private int outstandingReadLocks = 0;
    private ArrayList waitingForWriteLock = new ArrayList();
    private Thread writeLockedThread;
```

A `ReadWriteLock` object uses the above instance variables to keep track of threads that have requested or been issued a read or write lock. It uses the list referred to by the `waitingForWriteLock` variable to keep track of threads that are waiting to get a write lock. Using that list, it is able to ensure that write locks are issued in the same order that they are requested.

A `ReadWriteLock` object uses the `waitingForReadLock` variable to count the number of threads waiting to get a read lock. Simple counting is sufficient for this because all threads waiting for a read lock will be allowed to get them at the same time. That means that there is no reason to keep track of the order in which threads requested read locks.

A `ReadWriteLock` object uses the `outstandingReadLocks` variable to count the number of read locks that has issued but have not yet been relinquished by the threads they were issued to.

A `ReadWriteLock` object uses the `writeLockedThread` variable to refer to the thread that currently has a write lock. If no thread currently has a write lock from the `ReadWriteLock` object then the value of the `writeLockedThread` variable is null. By having a variable that refers to the thread that has been issued the write lock, the `ReadWriteLock` object can tell whether it awaked the thread to receive a write lock or the thread was awaked for another reason.

The `ReadWriteLock` class' `readLock` method appears below. It issues a read lock and returns immediately, unless there is an outstanding write lock. All that is does to issue a read lock is to increment the `outstandingReadLocks` variable.

```
synchronized public void readLock() throws InterruptedException {
        waitingForReadLock++;
        while (writeLockedThread != null) {
            wait();
        } // while
        waitingForReadLock--;
        outstandingReadLocks++;
    } // readLock()
```

A listing of the `writeLock` method appears below. The first thing you will notice is that it is longer than the `readLock` method. That is because it manages threads and a data structure. It begins by checking for the case in which there are no outstanding locks. If there are no outstanding locks, it issues a write lock immediately. Otherwise, it adds the current thread to a list that the `done` method uses as a queue. The current thread waits until it the `done` method issues it a write lock and then the `writeLock` method finishes by removing the current thread from the list.

```
public void writeLock() throws InterruptedException {
        Thread thisThread;
        synchronized (this) {
            if (writeLockedThread==null && outstandingReadLocks==0) {
                writeLockedThread = Thread.currentThread();
                return;
            } // if
            thisThread = Thread.currentThread();
```

```
            waitingForWriteLock.add(thisThread);
        } // synchronized(this)
        synchronized (thisThread) {
            while (thisThread != writeLockedThread) {
                thisThread.wait();
            } // while
        } // synchronized (thisThread)
        synchronized (this) {
            int i = waitingForWriteLock.indexOf(thisThread);
            waitingForWriteLock.remove(i);
        } // synchronized (this)
    } // writeLock
```

The final part of the `ReadWriteLock` class is the `done` method. Threads call a `ReadWriteLock` object's `done` method to relinquish a lock that the `ReadWriteLock` object previously issued to them. The `done` method considers three cases:

- There are outstanding read locks, which implies that there is no outstanding write lock.
  It relinquishes the read lock by decrementing the `outstandingReadLocks` variable. If there are no more outstanding read locks and threads are waiting to get a write lock, then it issues a write lock to the thread that has been waiting the longest to get a write lock. Then it wakes the waiting thread up.

- There is an outstanding write lock.
  It causes the current thread to relinquish the write lock. If there are any threads waiting to get the write lock, it transfers the write lock to the thread that has been waiting the longest by having the `writeLockedThread` refer to that thread instead of the current thread. If there are no threads waiting to get a write lock and there are threads waiting to get a read lock then it grants read locks to all of the threads that are waiting for a read lock.

- There are no outstanding locks.
  If there are no outstanding locks then the `done` method has been called at an inappropriate time, so it throws an `IllegalStateException`.

```
synchronized public void done() {
        if (outstandingReadLocks > 0) {
            outstandingReadLocks--;
            if ( outstandingReadLocks==0
                 && waitingForWriteLock.size()>0) {
                writeLockedThread = (Thread)waitingForWriteLock.get(0);
                writeLockedThread.notifyAll();
            } // if
        } else if (Thread.currentThread() == writeLockedThread) {
            if ( outstandingReadLocks==0
                 && waitingForWriteLock.size()>0) {
                writeLockedThread = (Thread)waitingForWriteLock.get(0);
                writeLockedThread.notifyAll();
            } else {
                writeLockedThread = null;
                if (waitingForReadLock > 0)
                  notifyAll();
            } // if
        } else {
            throw new IllegalStateException("Thread does not have lock");
        } // if
    } // done()
} // class ReadWriteLock
```

One last detail to notice about the `done` method is that is uses the `notifyAll` method, rather than the `notify` method. When it wants to allow read locks to be issued, it calls the `ReadWriteLock` object's `notifyAll` method to allow all of the threads waiting to get a read lock to proceed. When it issues the write lock to a thread, it calls that thread's `notifyAll` method. Calling its `notify` method will work in most cases. However, in the case that another thread is waiting to gain the synchronization lock of the thread to be issued the write lock, using the `notify` method could cause the wrong thread to wake up. Using the `notifyAll` method guarantees that the write thread will wake up.

Related Patterns

Single Threaded Execution

The Single Threaded Execution pattern is a good and simpler alternative to the Read/Write Lock pattern when most of the accesses to data are write accesses.

Scheduler

The Read/Write Lock pattern is a specialized version of the Scheduler pattern.

Producer-Consumer

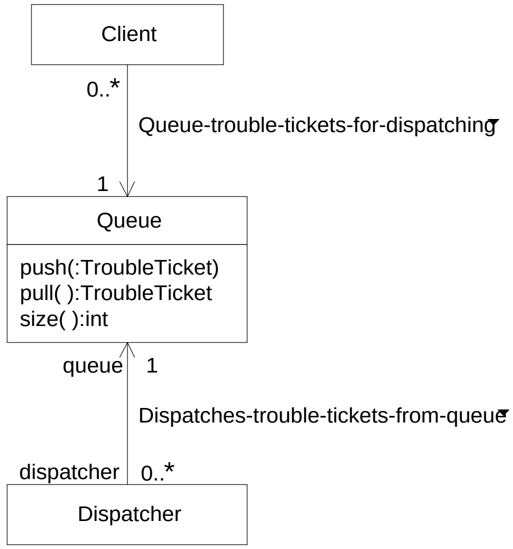
Synopsis

Coordinate the asynchronous production and consumption of information or objects.

Context

Suppose that you are involved in designing a trouble ticket dispatching system. Customers will enter trouble tickets through web pages. Dispatchers will review the trouble tickets and forward them to the person or organization best suited to resolve the problem.

Any number of people may be submitting trouble tickets through the web page at any given time. There will usually be multiple dispatchers on duty. When a trouble ticket comes in, if there are any dispatchers who are not busy, the system immediately gives the trouble ticket to one of them. Otherwise, it places the trouble ticket in a queue where the trouble ticket waits its turn to be seen by a dispatcher and dispatched. Below is a class diagram that shows a design for classes that implement that behavior.

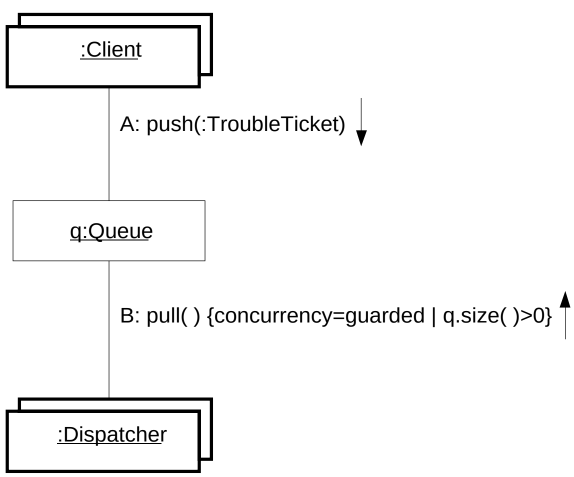


Trouble Ticket Classes

The above class diagram shows a `Client` class whose instances are responsible for getting trouble tickets filled out by users and placed in a `Queue` object. Trouble tickets stay in the `Queue` object until a `Dispatcher` object pulls them out of the `Queue` object.

The `Dispatcher` class is responsible for displaying trouble tickets to a dispatcher and then forwarding them to the destination selected by the dispatcher. When an instance of the `Dispatcher` class is not displaying a trouble ticket or forwarding it, it calls the `Queue` object's `pull` method to get another trouble ticket. If there are no trouble tickets in the `Queue` object, the `pull` method waits until it has a trouble ticket to return.

Below is a collaboration diagram that shows the interactions described above.



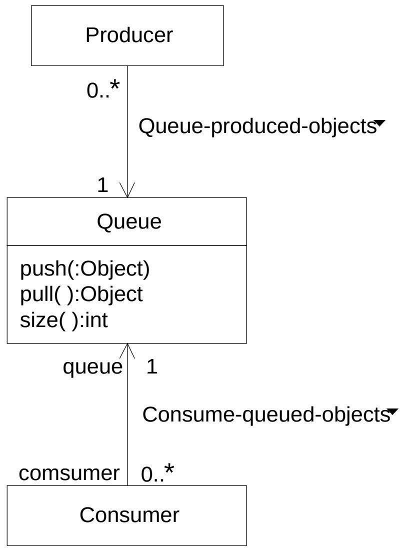
Trouble Ticket Collaboration

Forces

- Objects are produced or received asynchronously of their use or consumption.
- When an object is received or produced, there may not be any object available to use or consume it.

Solution

The class diagram below show the roles in which classes participate in the Produce-Consumer pattern.



Producer-Consumer Classes

Here are descriptions of the roles that classes can play in the Consumer-Producer pattern.

Producer

Instances of classes in this role supply objects that are used by `Consumer` objects. Instances of `Producer` classes produce objects asynchronously of the threads that consume them. That means that sometimes a `Producer` object will produce an object when all of the `Consumer` objects are busy processing other `Consumer` objects. Rather than wait for a `Consumer` object to become available, instances of `Producer` classes put the objects that they produce in a queue and then continue with whatever they do.

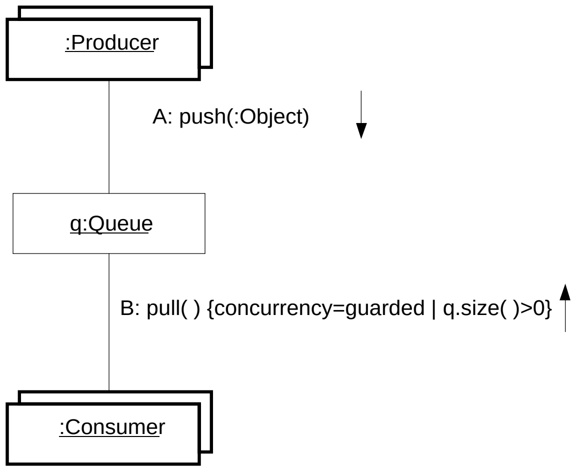
Queue

Instances of classes in this role act as a buffer for objects produced by instances of `Producer` classes. Instances of `Producer` classes place the objects that they produce in an instance of a `Queue` class. The objects remain there until a `Consumer` object pulls them out of the `Queue` object.

Consumer

Instances of `Consumer` classes use the objects produced by `Producer` objects. They get the objects that they use from a `Queue` object. If the `Queue` object is empty, a `Producer` object that wants to get an object from it must wait until a consumer object puts an object in the `Queue` object.

The collaboration diagram below shows the interactions between objects that participate in the Producer-Consumer pattern.



Producer-Consumer Collaboration

Consequences

- `Producer` objects are able to deliver the objects they produce to a `Queue` object without having to wait for a `Consumer` object.

- When there are objects in the `Queue` object, `Consumer` objects are able to pull an object out of the queue without waiting. However, when the queue is empty and a `Consumer` object calls the `Queue` object's `pull` method, the `pull` method does not return until a `Producer` object puts an object in the queue.

Implementation

In some implementations of the Producer-Consumer pattern, the queue cannot grow beyond a maximum size. If that is the case, it implies that there is a special case to consider when the queue is at its maximum size and a producer thread wants to put an object in the queue. The usual way to handle that is for the queue to use the Guarded Suspension pattern to force the producer thread to wait until a consumer thread has removed an object from the queue. When there is room in the queue for the object that the producer wants to put in the queue, the producer thread is allowed to finish putting the object in the queue and proceed with whatever else it does.

JAVA API Usage

The core Java API includes the classes `java.io.PipedInputStream` and `java.io.PipedOutputStream`. Together, they implement a variant of the Producer-Consumer pattern called the Pipe pattern. The Pipe pattern only involves one `Producer` object an only one `Consumer` object. The Pipe pattern usually refers to the `Producer` object as a data source and the `Consumer` object as a data sink.

The `java.io.PipedInputStream` and `java.io.PipedOutputStream`. classes jointly fill the role of `Queue` class. They allow one thread to write a stream of bytes to one another thread. The threads are able to perform their writes and read asynchronously of each other, unless the internal buffer that they use is empty or full.

Code Example

The listings below show code the implements the design discussed under the "Context" heading: The first two listings shown are skeletal listings of the `Client` and `Dispatcher` classes.

```
public class Client implements Runnable {
```

```
    private Queue queue;
    //...
    public void run() {
        TroubleTicket tkt = null;
        //...
        queue.push(tkt);
    } // run()
} // class Client


public class Dispatcher implements Runnable {
    private Queue queue;
    //...
    public void run() {
        TroubleTicket tkt = queue.pull();
        //...
    } // run()
} // class Dispatcher
```

The last listing is of the Queue class.

```
public class Queue {
    private ArrayList data = new ArrayList();

    /**
     * Put an object on the end of the queue
     * @param obj the object to put at end of queue
     */
    synchronized public void push(TroubleTicket tkt) {
        data.add(tkt);
        notify();
    } // put(TroubleTicket)

    /**
     * Get an TroubleTicket from the front of the queue
     * If queue is empty, wait until it is not empty.
     */
    synchronized public TroubleTicket pull() {
        while (data.size() == 0){
            try {
                wait();
            } catch (InterruptedException e) {
            } // try
        } // while
        TroubleTicket tkt = (TroubleTicket)data.get(0);
        data.remove(0);
        return tkt;
    } // get()

    /**
     * Return the number of trouble tickets in this queue.
     */
    public int size() {
        return data.size();
    } // size()
} // class Queue
```

Related Patterns

Guarded Suspension

The Producer-Consumer pattern uses the Guarded Suspension pattern to manage the situation of a `Consumer` object wanting to get an object from an empty queue.

Pipe

The Pipe pattern is a special case of the Producer-Consumer pattern that involves only one `Producer` object an only one `Consumer` object. The Pipe pattern usually refers to the `Producer` object as a data source and the `Consumer` object as a data sink.

Scheduler

The Producer-Consumer pattern can be viewed as a special form of the Scheduler pattern that has scheduling policy with two notable features.

- The scheduling policy is based on the availability of a resource.
- The scheduler assigns the resource to a thread but does not need to regain control of the resource when the thread is done so it can reassign the resource to another thread.

Two Phase Termination [Grand98]

Synopsis

Provide for the orderly shutdown of a thread or process through the setting of a latch. The thread or process checks the value of the latch at strategic points in its execution.

Context

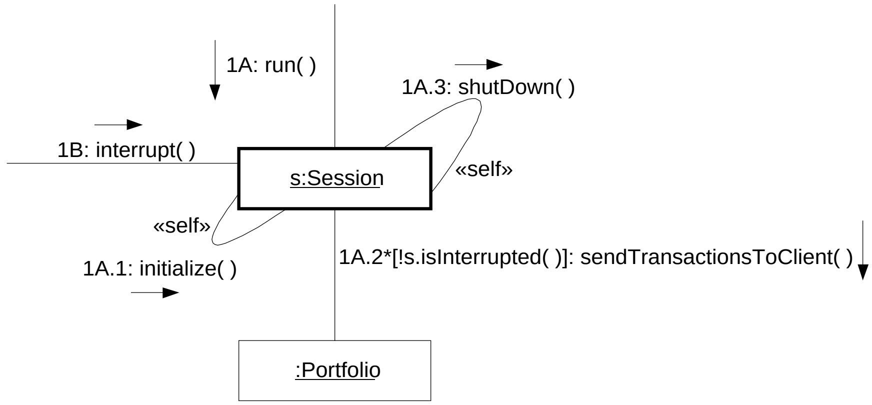
Suppose that you are responsible for writing a server that provides middle tier logic for a stock trading workstation. A client connects to the server. The client then indicates that it is interested in certain stocks. The server sends the current price of those stocks to the client. When the server receives information that shares of a stock have been traded, it reports the trade to clients who are interested in the stock.

Part of the internal mechanism that the server uses to provide that service is to create a thread for each client. That thread is responsible for delivering information about stock trades to the client that it serves.

Aside from its core functions, there are some administrative commands that the server must respond to. One of those commands is a command that forces the disconnection of a client. When the server is asked to disconnect a client, from an internal point of view that means shutting down the thread that is servicing that client and the releasing the related set of resources that thread is using.

Another administrative command that the server must respond to is a command to shut down the entire server.

Both commands are similar in what they do. The main difference is their scope. One command shuts down a single thread. The other command shuts down an entire process. In both cases, the implementation techniques are similar. Below is a collaboration diagram that show how a server thread could be organized to shut down cleanly on request.



Server Thread Shut Down

The above collaboration begins with a call to a `Session` object's `run` method. The `run` method first calls the `Session` object's `initialize` method. It then repeatedly calls the `Portfolio` object's `sendTransactionsToClient` method. It keeps calling that method as long as the `Session` object's `isInterrupted` method returns false. The `Session` object's `isInterrupted` method returns false until the `Session` object's `interrupt` method is called.

The normal sequence of events that shuts down a session begins with a different thread than the one that called the `run` method. That other thread calls the `Session` object's `interrupt` method. The next time that the `Session` object calls its `run` method, it returns true. The `run` method then stops calling the `Portfolio` object's `sendTransactionsToClient` method. It then calls the `Session` object's `shutDown` method, which performs any necessary cleanup operations.

The technique for shutting down a process in an orderly manner is similar to the technique for threads. When a command is received to shut down an entire process, it sets a latch that causes every thread in the process to shut down.

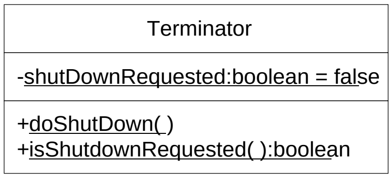
Forces

- A thread or process is designed to run indefinitely.
- There could be unfortunately results if a thread or process is forcibly shut down without first having the chance to clean up after itself.
- When a thread or process is asked to shut down, it is acceptable for the thread or process to take a reasonable amount of time to clean up before shutting down.

Solution

There are two basic techniques for shutting down a thread or a process. One way is to immediately terminate them. The other way is to ask the thread or process to terminate and then expect it to comply with the request by first performing any necessary clean up and then terminating.

The following class diagram shows a class that you might use to coordinate the shutdown of a process:



Process Shutdown

A class, such as the one above, is used to allow a process to receive a request to shut down and then clean up after itself before shutting down. Each of the process' threads call the `isShutdownRequested` method at strategic points in their execution. If it returns true, the thread cleans up and shuts down. When all the application threads have died, the process exits.

Shutting down an individual thread involves using a thread specific latch. Every Java thread has one because it is part of the `Thread` class. It is set to true by calling the thread's `interrupt` method. It is queried by calling the thread's `isInterrupted` method.

Consequences

- Using the Two Phase Termination pattern allows processes and threads to clean up after themselves before they terminate.

- Using the Two Phase Termination pattern can delay the termination of a process or thread for an unpredictable amount of time.

Implementation

After a process or thread has been requested to shut down, it can be difficult or impossible to determine if the process or thread will actually terminate until it does. For that reason, if there is any uncertainty that the thread or process will terminate after it has been requested to do so, after a predetermined amount of time the thread or process should be forcibly terminated.

A thread can be forcibly terminated by calling its `stop` method. The mechanism for forcible terminating a process varies with the operating system.

Methods that set a termination latch to true do not need to be synchronized. Flag setting is idempotent. The operation can be performed by one or more threads, concurrently or not, and the result is still that the termination latch is set to true.

JAVA API Usage

The core Java API does not make use of the Two Phase Termination pattern. However, it does have features to support the Two Phase Termination pattern.

To support the two phase termination of threads, the `Thread` class provides the `interrupt` method to request a thread's termination. `Thread` class also provides the `isInterrupted` method to allow a thread to find out if its termination has been requested.

There are some methods that such as `sleep` that are known to put a thread in a wait state. There is an assumption that if a thread is asked to shut down while its is waiting for one of those methods to return, the thread will detect the request for its termination and comply as soon as the waiting method to returns. To help insure the timeliness of a thread's shutdown, some methods that cause a thread to wait for something throw an

`InterruptedException` if a thread is waiting for one of those methods to return when its `interrupt` method is called. The methods that can throw an `InterruptedException` include `Thread.sleep`, `Thread.join` and `Object.wait`. There are a number of others.

To support the shutdown of a process, when a thread dies, if there are no threads still alive that are not daemon threads then the process shuts down.

Note that Java does not provide any direct way of detecting or catching signals or interrupts from an operating system that cause a process to shut down.

Code Example

The listing below shows code that implements the design discussed under the "Context" heading.

```
public class Session implements Runnable {
    private Thread myThread;
    private Portfolio portfolio;
    private Socket mySocket;
...
    public Session(Socket s) {
        myThread = new Thread(this);
        mySocket = s;
        ...
    } // constructor()

    public void run() {
        initialize();
        while (!myThread.interrupted()) {
            portfolio.sendTransactionsToClient(mySocket);
        } // while
        shutDown();
    } // run()

    /**
     * Request that this session terminate.
     */
    public void interrupt() {
        myThread.interrupt();
    } // interrupt()

    /**
     * Initialize ths object.
     */
    private void initialize() {
        //...
    } // initialize()

    /**
     * perform cleanup for this object.
     */
    private void shutDown() {
        //...
    } // shutDown()
...
} // class Session
```