Executive Summary: Balking Design Patterns By Drew Goldberg

Balking Patterns are used to prevent an object from executing certain code if it is an incomplete or inappropriate state.

These Design Patterns include: The Balking Pattern, The Guarded Suspension, and The Double-Checked Locking Pattern.

These patterns seem to have arisen when JVMs were slower and synchronization wasn't as well understood as it is today.

These patterns have appeared become somewhat antiquated as JVMs improved, newer design pattern(s) were introduced, and the increased use of functional programming language's immutable/persistent data structures help reduce the complexities of concurrency's implementation and maintenance.

Still it is nice to see what was used in the past inorder to get a better perspective of today's concurrency techniques.

Balking - Design Patterns

Dealing with Incomplete and Incorrect States

By

Drew Goldberg

What are balking patterns?

- Balking - If an object's method is invoked when the object is in an inappropriate state, then the method will return without doing anything.

  Balking Design Patterns:

- Balking Design Pattern
- Guarded Suspension
- Double Checked Locking

Reference: http://www.mindspring.com/~mgrand/pattern_synopses.htm#Balking

Balking Pattern: Intro

- This software design pattern is used to invoke an action on an object only when the object is in a particular state.

- Objects that use this pattern are generally only in a state that is prone to balking temporarily but for an **unknown** amount of time

  reference: http://en.wikipedia.org/wiki/Balking_pattern

Balking Pattern: Implementation

```
public class Example {
   private boolean jobInProgress = false;

   public void job() {
      synchronized(this) {
         if (jobInProgress) {
            return;
         }
         jobInProgress = true;
      }
      // Code to execute job goes here
      // ...
   }

   void jobCompleted() {
      synchronized(this) {
         jobInProgress = false;
      }
   }
}
```

If the boolean instance variable jobInProgress is set to false, then the job() will return without having executed any commands and therefore keeping the object's state the same.

If jobInProgress variable is set to true, then the Example object is in the correct state to execute additional code in the job()

reference: http://en.wikipedia.org/wiki/Balking_pattern

Balking Pattern and Single Threaded Execution

Typically, a balking pattern is used with a single threaded execution pattern to help coordinate an object's change in state.

reference: http://www.mindspring.com/~mgrand/pattern_synopses.htg

What is the Single Threaded Execution pattern?

- This design pattern describes a solution for the concurrency when multiple readers and multiple writers access to a single resource.
- The most common problems in this situation were lost updates and inconsistent reads.
- Essentially, Single Threaded Execution is a concurrency design pattern that implements a form of shared mutability, which is still very easy to screw up.

reference: Checking Java Concurrency Design Patterns Using Bandera Cleidson R. B. de Souza and Roberto S. Silva Filho Department of Information and Computer ScienceUniversity of California, Irvine

http://wendang.baidu.com/view/df10870a581b6bd97f19eafb.html?from=related

Balking Pattern: Negatives

- It is considered an anti-pattern, so it is not a true design pattern

- Since, the balking pattern is typically used when an object's state could be prone to balking for an indefinite period of time, then it **isn't recommended** to use when an object's state is prone to balking for a **relative known amount of time**.

- The Guarded Suspension pattern is a good alternative when an object's state is prone to balking for a known **finite period of time**.

  reference: http://en.wikipedia.org/wiki/Balking_pattern

Guarded Suspension Pattern: When To Use It

- Both Guarded Suspension Pattern and the Balking Pattern use similar criteria
- It manages operations that require both a lock to be acquired and a precondition to be satisfied before the operations can be executed.
- The guarded suspension pattern is typically applied to method calls in object-oriented programs
- It involves suspending the method call and the calling thread, until the precondition is satisfied.
- The amount of time for the preconditioned to be satisfied is usually a relatively known amount of time

reference: http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded Suspension Pattern Intro:

- This design pattern uses try/catch clauses because an InterruptedException can be thrown when the wait() is invoked.

- The wait() method is called in the try clause if the precondition isn't met

- The notify()/notifyAll() method is called to update a single/all other threads that something has happened to the object

.

- The notify()/notifyAll() are usually used for telling the other threads that the object's state has been changed.

reference: http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded Suspension Pattern: Bad Code, That Can Use This Pattern

CODE THAT CAN USE THE GUARDED SUSPENSION PATTERN

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

What's wrong with this code?

Suppose, for example guardedJoy() is a method that must not proceed until a shared variable joy has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied. This can wastes a lot CPU cycles!

Reference http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded Suspension Pattern: How to Fix This Code

```
public synchronized guardedJoy() {
      while(!joy) {
        try {
                wait();
        } catch (InterruptedException e) {}
      }
      System.out.println("Joy and efficiency have been achieved!");
}

public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

Instead of looping over a variable to see if the precondition of joy equals true has been met, use a Guarded Block.

The wait() invocation blocks the thread until it receives a notified response.

If the precondition joy then equals, true, it can continue executing the code.

Reference: http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded Suspension Pattern: Implementing a Guarded Block

- A more efficient guard invokes Object.wait to suspend the current thread. The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

- This blocks the thread, preventing it from further executing code until it receives the notification to proceed.

reference: http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded Suspension Pattern: Brief Synopsis wait(), notify(), notifyAll()

- Wait, Notify, and NotifyAll are final methods of the Object class.
- wait() is an overloaded function that comes in three varities: wait(), wait(long timeout, int nanos), wait(long timeout). This allows you to specify how long the thread is willing to wait.
- Invoking these methods cause the current thread (call it T) to place itself in a wait state for this object and then to relinquish any and all synchronization claims on this object. Thread T becomes disabled for thread scheduling purposes and lies dormant until one of four things happens:
    - Some other thread invokes the notify method for this object and thread T happens to be arbitrarily chosen as the thread to be awakened.
    - Some other thread invokes the notifyAll() method for this object.
    - Some other thread interrupts thread T.
    - The specified amount of real time has elapsed, more or less. If timeout is zero, however, then real time is not taken into consideration and the thread simply waits until notified.

reference: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Guarded Suspension Pattern: notify(), notifyAll()

- notify() - Wakes up a single thread that is waiting for the object's monitor lock.
- If more than one thread is waiting for the object's monitor lock, the notify method wakes up an arbitrary thread that is waiting on the object's monitor lock.
- notifyAll() - Wakes up all threads waiting on the object's monitor lock

  reference: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Guarded Design Pattern: Negatives

- Because it is blocking, the guarded suspension pattern is generally only used when the developer knows that a method call will be suspended for a finite and reasonable period of time. If the blocking can be for an indefinite amount of time, use the Balking Design Pattern

- If a method call is suspended for too long, then the overall program will slow down or stop, waiting for the precondition to be satisfied.

reference: http://en.wikipedia.org/wiki/Guarded_suspension

Guarded Design Pattern: Negatives Continued

- If multiple threads are waiting to access the same method, the Guarded Design Pattern doesn't pick which thread will execute the method next.

- Thread notification isn't under the programmers' control

- In order to be able to pick which thread will be executed next, use the Scheduler design pattern instead.

  reference: Multi-Thread Design Patterns by Mark Grand, Clickblocks LLC
  http://www.ajug.org/meetings/download/Multi-Threading_Design_Patterns.pdf

Guarded Design Pattern: Negatives Continued

The Scheduler Design Pattern

It controls the order of when waiting threads can execute single threaded code in a multi-threaded program.

Disadvantages:

Adds lot of overhead on when a synchronized method can be executed.

references: http://en.wikipedia.org/wiki/Scheduler_pattern

Guarded Design Patten Negatives: Continued

- Always invoke wait() inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true because the notify isn't thread specific.

refernce: http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

Guarded Suspension: Related Patterns

The Two-Phase Termination design pattern can be used with the Guarded Suspension design pattern.

reference: Multi-Thread Design Patterns by Mark Grand, Clickblocks LLC
http://www.ajug.org/meetings/download/Multi-Threading_Design_Patterns.pdf

Double-Checked Locking: Introduction

- Is a software design pattern used to reduce the overhead of acquiring a lock which was more significant in the past.
- It first testing the locking condition without actually acquiring the lock. If the first test for locking passes, then the actual locking implementation occurs.
- It uses the technique of lazy initialization

reference: http://en.wikipedia.org/wiki/Double-checked_locking

Double-Checked Locking: Lazy Initialization

- Why use lazy initialization - because the performance of early JVMs wasn't very good.

  Better yet, what is lazy initialization?

- Lazy initialization delays the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.

  reference - http://en.wikipedia.org/wiki/Lazy_initialization
  reference: Java Concurrency In Practice by Brian Goetz, pg. 348.

Double-Checked Locking: Lazy Initialization Example

Example of Lazy Initialization using the factory method pattern in java.

```
public class Fruit {
      private String typeName;
      private static Map<String, Fruit> types = new HashMap<String, Fruit>();
      private Fruit(String typeName) {
            this.typeName = typeName;
      }

            public static Fruit getFruitByTypeName(String type) {
             Fruit fruit;

             if (!types.containsKey(type)) {
                   // Lazy initialization
                   fruit = new Fruit(type);

                   types.put(type, fruit);
             } else {
                   // Okay, it's available currently
                   fruit = types.get(type);
             }

             return fruit;
      }
```

Here is the lazy initialization of the Fruit object. If the String type parameter provided in the function, isn't a key, then a new fruit object is created, else the function returns an already existing fruit type.
reference: http://en.wikipedia.org/wiki/Lazy_initialization

Double-Checked Locking: How It Works

- It checks whether or not you need to initialize without synchronizing, lazy initialization.
- If the object it is looking for isn't null then use it.
- Else if it is null, synchronize and check again, to make sure the resource isn't initialized and then allow only one thread to initialize it.

reference: Java Concurrency In Practice by Brian Goetz, pg. 348.

Double-Checked Locking: Implementation Example

Implementation Double-Checked Locking to the lazy initialization example

```
public static Fruit getFruitByTypeNameHighConcurrentVersion(String type) {
    Fruit fruit;

    if (!types.containsKey(type)) {
        synchronized (types) {
            if (!types.containsKey(type)) {
                // Lazy initialization
                types.put(type, new Fruit(type));
            }
        }
    }

    fruit = types.get(type);

    return fruit;
}
```

Make sure that after acquiring the lock on the types object that no other thread has created a fruit object. If not, then use lazy initialization to create a new Fruit object.

reference: http://en.wikipedia.org/wiki/Lazy_initialization

Double-Checked Locking: What can go wrong

How can Double-Checked Locking fail Prior to Java 5.0

- The problem with double-checked locking is that there is no guarantee it will work on single or multi-processor machines.
- The issue of the failure of double-checked locking is not due to implementation bugs in JVMs but to the older Java platform memory model.
- The memory model allowed "out-of-order writes" to occur and is why Double-Checked Locking fell out of favor..

reference: http://www.ibm.com/developerworks/java/library/j-dcl/index.html

Double-Check Locking: Failing Due To Out of Order Memory Writes

A classic example of a partially initialized object escaping during its construction.

```
public class PieChucker {
        // Singleton instance, lazily initialized
        private static PieChucker instance;
        public static PieChucker getInstance() {
                if(instance == null) {
                        synchronized(PieChucker.class) {
                                if(instance == null) {
                                        instance = new PieChucker();
                                }
                        }
                }
        return instance;
}

        // Private as it's only constructed by getInstance()
        private PieChucker() {
        }

        public void fling(Target target) {
        // … chuck pie at target
        }
}
```

Reference: http://tech.puredanger.com/2007/06/15/double-checked-locking/

Consider this scenario:
Thread 1 enters getInstance(), sees null on both if checks, and starts constructing the singleton.

Thread 2 enters getInstance() and checks the first if check.

At this point, thread 2 has not yet entered the synchronized block, so has not established a "happens-before" relationship with Thread 1.

It is thus undefined whether Thread 2 will see a fully constructed instance (you got lucky!), a partially constructed instance (probably real bad), or null (causing two singletons to get constructed and breaking the singelton-ness).

Double-Check Locking: Solutions

Solutions to the Broken Double-Checked Locking pattern:

- Synchronized locking – Lazy Initialization
- Double-checked locking with volatile – Lazy Initialization
- Static initialization – Eager Initialization

reference: http://tech.puredanger.com/2007/06/15/double-checked-locking/

Double-Checked Locking: Solutions Since Java 5.0

Java 5.0 added the volatile keyword was specifically changed to ensure that Double-Checked Locking is safe.

reference: http://www.javamex.com/tutorials/double_checked_locking_fixing.shtml

Conclusion

Overall balking design patterns seem either obsolete or possibly dangerous to use.

The Guarded Suspension Method seems antiquated when compared to the scheduler design pattern which provides finer granularity for thread notification.

Conclusion Continue

The Double-Check Locking design pattern was once very useful when the overhead for synchronizing on an object was very expensive.

With modern JVMs, the overhead cost for synchronizing isn't as detrimental to the program.

Conclusion Continued

Also, there doesn't seem to be consensus regarding whether or not this design pattern prevents out of order memory writes due to an object's ability to escape during its construction.

Conclusion Continued

Another reason for balking patterns becoming obsolete is the use of functional programming languages like Clojure, Groovy, and other languages/libraries that incorporate Software Transactional Memory (STM).

STM makes an object's state immutable which mitigates concerns of it being in an incorrect state during concurrent operations.

References

1. "Overview of Design Patterns." *Design Pattern Synopses*. Web. <http://www.mindspring.com/~mgrand/pattern_synopses.htm>.
2. "Balking Pattern." *Wikipedia*. Wikimedia Foundation. Web. <http://en.wikipedia.org/wiki/Balking_pattern>.
3. http://wendang.baidu.com/view/df10870a581b6bd97f19eafb.html?from=related
4. "Guarded Blocks." *(The Javaâ„¢ Tutorials Essential Classes Concurrency)*. Web. <http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html>.
5. "Object (Java Platform SE 7)." *Oracle Documentation*. Web. <http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>.
6. "Guarded Suspension." *Wikipedia*. Wikimedia Foundation. Web. <http://en.wikipedia.org/wiki/Guarded_suspension>.
7. http://www.ajug.org/meetings/download/Multi-Threading_Design_Patterns.pdf
8. "Scheduler Pattern." *Wikipedia*. Wikimedia Foundation. Web. <http://en.wikipedia.org/wiki/Scheduler_pattern>.
9. "Double-checked Locking." *Wikipedia*. Wikimedia Foundation. Web. <http://en.wikipedia.org/wiki/Double-checked_locking>.
10. "Lazy Initialization." *Wikipedia*. Wikimedia Foundation. Web. <http://en.wikipedia.org/wiki/Lazy_initialization>.
11. "Double-checked Locking and the Singleton Pattern." *302 Found*. Web. <http://www.ibm.com/developerworks/java/library/j-dcl/index.html>.
12. "One More Look at Double-Checked Locking." *: Pure Danger Tech*. Web. <http://tech.puredanger.com/2007/06/15/double-checked-locking/>.
13. "Double-checked Locking (DCL) and How to Fix It (ctd)." *Double-checked Locking*. Web. <http://www.javamex.com/tutorials/double_checked_locking_fixing.shtml>.
14. Goetz, Brian. *Java Concurrency in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2006. Print.