JOURNAL OF OBJECT TECHNOLOGY

. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6. No. 3, March-April 2007

The Parametric Singleton Design Pattern

Douglas Lyon and **Francisco Castellanos**

ABSTRACT

The parametric singleton design pattern combines the singleton design pattern with a parameter that enables unique creation of instances of a class. These instances are cached in a table. When a user asks for an instance with these parameters, the table is checked and instances are created conditionally.

Parametric lazy instantiation causes instance creation, with the given parameters, if, and only if, it is not already in the table. Thus, the table yields the instance with optional creational effort. Lazy instantiation is not new, nor, for that matter, is the singleton design pattern. However, parametric lazy instantiation is new and so is the parametric singleton.

We apply our parametric singleton design pattern to the retrieval of RMI registries bound to a given port. The goal of our system is to make sure that no two RMI registries on the same machine are listening to the same socket and to make use of the RMI registries after creation. RMI registries are used in distributed computation.

Other applications include parametric resources that are countable in number, locked, when in use and released when finished. Examples include file systems, serial ports, parallel ports, video cameras, microphones and audio output streams.

1 INTRODUCTION

The singleton design pattern ensures that only a single instance of a class exists. It is also meant to provide a global point of access to it.

In comparison, the intent of the parametric singleton design pattern is to ensure that a class has only one instance for *a given set of parameter values*. Like the singleton design pattern, it also provides a global point of access to it.

In the example that follows, we will show that the number of resources is not known, in advance. We will show that it is possible to have several registries on several ports. Further, we demonstrate that the singleton design pattern implementation is responsible for keeping track of them all.

2 MOTIVATION

A system cannot tolerate multiple instances of some classes with identical parameters. For example, you cannot have two instances making use of the same serial port, at the same time. You cannot have two instances that are trying to listen to the same socket connection. You cannot have two instances writing to the same file structure at the same time.

Operating systems often have the role as arbiter of consumed resources (i.e., tape drives, serial ports, etc.) we frequently leave it to the operating system to resolve these contention issues.

Even for similar resources (like serial ports) there are few standards for handling contention. For example, serial ports on Linux are locked using a *"/var/lock"* file. POSIX, on the other hand, uses *ioctl* mechanisms to make sure that no other process will open the port. Also, lock files depend on inconsistent naming conventions.

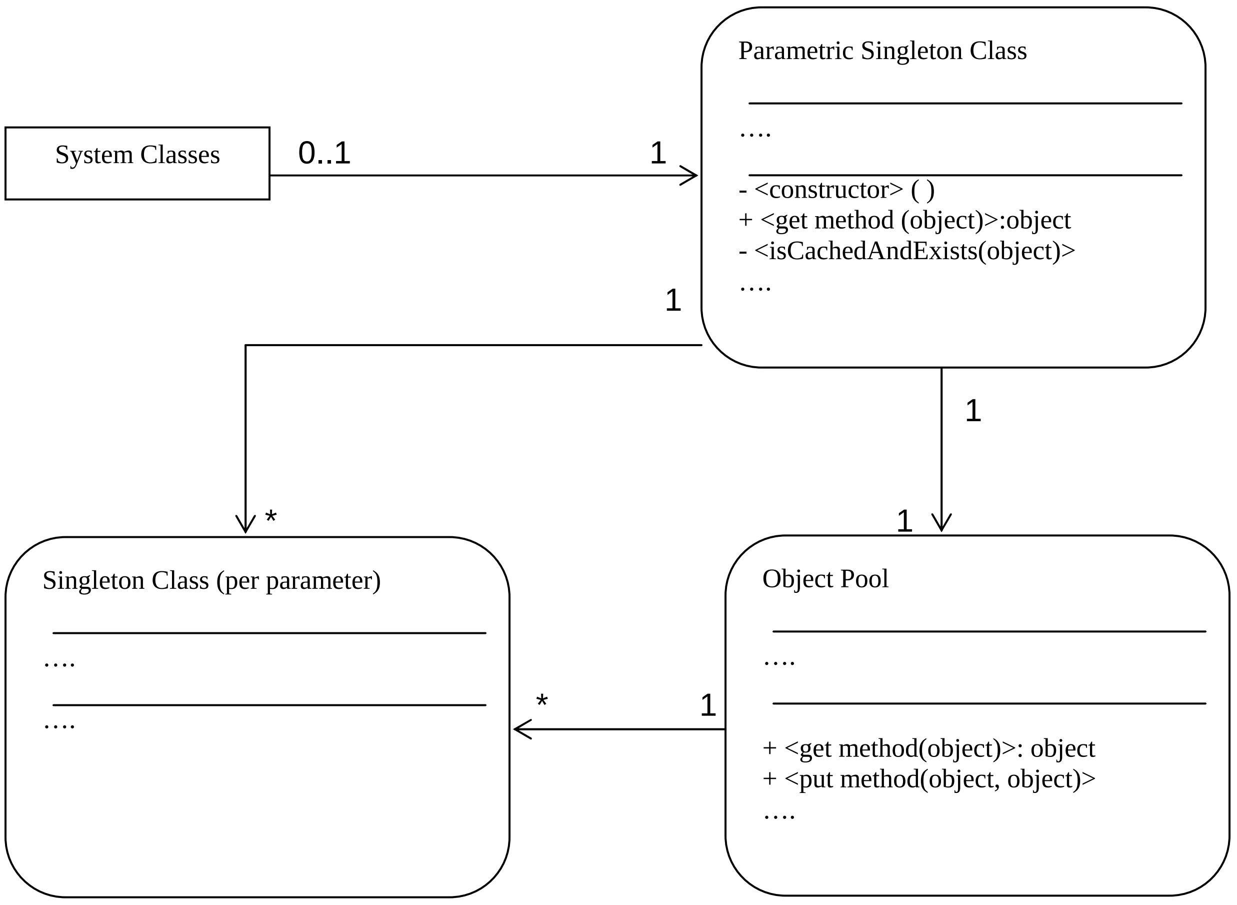
A better solution is to make a class that is responsible for keeping track of the instances created from the class. The class is declared final, so that it cannot be sub classed. The class also has a private constructor, so that other classes cannot instance it. The new design pattern is called the parametric singleton design pattern and it provides a way to access and create instances with given parameters.

3 APPLICABILITY

Use the Parametric Singleton Design Pattern when:

1. There must be exactly one instance of a class with the given parameters.
2. The instances must be accessible to clients from a well-known access point.

4 STRUCTURE



Parametric Singleton Class
….
- <constructor> ( )
+ <get method (object)>:object
- <isCachedAndExists(object)>
….
System Classes
0..1
1
1
1
*
1
Singleton Class (per parameter)
….
….
Object Pool
….
+ <get method(object)>: object
+ <put method(object, object)>
….
*
1

Figure 4.1 A Class diagram for the parametric singleton pattern.

5 PARTICIPANTS

The participants are Parametric Singleton Clients that need instances.

1. The Parametric Singleton defines an instance upon request from a client, if, and only if, the instance does not already exist.
2. The Parametric Singleton returns the instance to the client.
3. The Parametric Singleton is responsible for creating unique instances from given parameters.

6 COLLABORATIONS

Clients obtain a reference to a Parametric Singleton instance only through the parametric singleton. If the instance is left in an improper state (e.g., the serial port was left open) it is NOT the role of the Parametric Singleton Design Pattern to close the IO port. Nor is it the role of the Parametric Singleton Design Pattern to open the port. That role is delegated to another part of the system.

Further, it is not the role of the Parametric Singleton Design Pattern to check out resources. That is, multiple threads can have multiple references to the same resource at the same time. MUTEX locking is delegated to some other part of the system.

7 CONSEQUENCES

The Parametric Singleton Design Pattern has several benefits:

1. The parametric singleton design pattern controls creation of parametrically defined instances. The Parametric Singleton class uses parametric lazy instantiation to make new instances.
2. Reduced name space. The Parametric Singleton pattern avoids global variables that store instances created from the same parameter.

8 IMPLEMENTATION

Here are implementation issues to consider when using the Parametric Singleton pattern:

1. Unique mapping of parameters. The Parametric Singleton pattern requires that there be a means to isomorphically map the parameter space into the instance space, as shown in Example 8.1.

Example 8.1

```
/**
* Single instance of the registry is instances per port. The instance
* is put into a hash table.
*/
private static Hashtable registry = new Hashtable(100);
…
private static void startRegistry(Integer port) throws RemoteException
         {
  Registry r = null;
  try {
      r = LocateRegistry.createRegistry(port.intValue());
      registry.put(port, r);
        } catch (Exception e) {
        r =LocateRegistry.getRegistry(port.intValue());
            registry.put(port, r);
        }
    }
```

2. Ensure unique instances. The Parametric Singleton pattern makes unique instances from a unique parameter, as shown in Example 8.2.

Example 8.2

```
/**
* Checks whether a Registry has been started on this host for the
        specified port.
* Otherwise it starts one.
*/
   public static void restart(Integer port) {
        if (isInCacheAndRunning(port))
            return;
        try {
            startRegistry(port);
        } catch (RemoteException e) {
            In.message(e);
        }
    }
```

3. Cache instances for fast retrieval. The Parametric Singleton pattern must be able to look up instances, given some set of parameters, and do so from some data structure. That is, there must be enough space to hold references to all the instances the program will need. Also, a mechanism is needed to look up and retrieve the instances quickly enough to satisfy the clients, as shown in example 8.3.

Example 8.3

```
private static Hashtable registry = new Hashtable(100);

private static void startRegistry(Integer port) throws RemoteException
        {
      Registry r = null
       …
      r =LocateRegistry.getRegistry(port.intValue());
      registry.put(port, r);
       …
}

public static Registry getRegistry(Integer port) {
        restart(port);
        return (Registry)registry.get(port);
}
```

9 SAMPLE CODE

The following code uses the parametric singleton pattern to create one, and only one instance of the RMI registry per port. A registry request causes a cache search. If no registry is found on the given port, a new one is created. Thus we make use of lazy instantiation.

```
public final class ParametricSingletonRmiRegistry {

     private static Hashtable registry = new Hashtable(100);
     // use singleton pattern and prevent instantiation of
        RmiRegistryUtils.

     private ParametricSingletonRmiRegistry() {
     }
     // the only instance of the Registry held by the
        RmiRegistryUtils

    public static void listNames(Integer port) {
         if (!isInCacheAndRunning(port))
             return;
         Registry r = (Registry) registry.get(port);
         System.out.println("registry on port: " + port);
         System.out.println("registry: " + r);
         System.out.println("names: ");
         try {
             print(r.list());
         } catch (RemoteException e) {
             In.message(e);
         }
         System.out.println("------------------------");
     }

  /**
      * Checks whether a Registry has been started on this
        host
      * in the specified port. If not, it tries to start
        one.
      */
     public static void restart(int port) {
         if (isInCacheAndRunning(port))
             return;
         try {
             startRegistry(port);
         } catch (RemoteException e) {
             In.message(e);
         }
     }
    /**
      * @param port
      * @return true if registry is in cache and
        running
      */
     private static boolean isInCacheAndRunning(Integer port) {
         if (!isRegistryRunning(port.intValue())) return false;
         if (!isRegistryInCache(port.intValue())) return false;
         return true;
     }

 /**
* @param port
* @return true if we are able to locate a local registry
*/
     private static boolean isRegistryRunning(int port) {
         try {
             Registry r = LocateRegistry.getRegistry(port);
             if (r != null) return true;
         } catch (RemoteException e) {
             return false;
         }
         return false;
     }

     /**
      * @param port
      * @return return true if registry is in
        cache
      */
     private static boolean isRegistryInCache(int port) {
         if (registry.get(new Integer(port)) == null) return false;
         return true;
     }

   /**
      * Restart the registry, if
        needed.
      * Return the only instance of the registry for consistent
        global
      * using by making use of the Singleton Design
        Pattern
      *
      * @return the internally held registry
        instance.
      */
     public static Registry getRegistry(Integer port) {
         restart(port);
         return (Registry) registry.get(port);
     }
     private static void startRegistry(Integer port) throws
        RemoteException {
         Registry r;
         try {
             r = LocateRegistry.createRegistry(port.intValue());
             registry.put(port, r);

         } catch (Exception e) {
             r = LocateRegistry.getRegistry(port.intValue());
             registry.put(port, r);
         }
     }
    public static void main(String args[]) throws RemoteException {
         //restart(port);
         restart(5001);
         restart(5002);
         restart(5002);
         Registry r = LocateRegistry.getRegistry(5002);
         System.out.println("registry="+r);
}
}
```

10 RELATED WORK

The singleton design pattern is not new [Gamma et Al.]. However, the parametric singleton design pattern is new, as far as we know. While both the singleton and factory design patterns are creational patterns, the singleton ensures that the instance you get is the sole instance [Goldfedder]. In comparison, the parametric singleton ensures that the instance you get is the sole instance that corresponds to the given parameters.

There are other extensions to the singleton design pattern, such as *Optional Singletons. However*, the optional singleton creates new behavior for the subclass, rather than limits the behavior, like the parametric singleton [Maclean].

The idea that the singleton pattern implementation is awkward to subclass is not new. The declaration of the class as final has therefore become common. The parametric singleton pattern is just like the cache management pattern, with a difference; the cache management pattern does not limit the number of instances with a given id. The cache management design helps to speed instance creation. In fact, the pattern has many similarities with the *object pool* design pattern. Primary differences are instance homogeneity and a check-in mechanism. The parametric singleton has no such restriction [Grand].

The use of the lazy initialization design pattern is not new, and, in fact, is a practical requirement when dealing with an unknown number of instances in the cache [Beck97].

The use of synchronized in the singleton design pattern is not new either, typically called the *double-checked locking singleton design pattern*. This is a thread-safe means of implementing the parametric singleton design pattern [Shalloway]. The synchronized keyword ensures that method invocations will be atomic (i.e., they will happen all at once). If this were not the case, a race-condition would occur between two threads requesting the same object with the same parametric arguments at the same time and this leads to unpredictable behavior.

11 SUMMARY

The parametric singleton pattern allows for one instance of a class for a given set of parameters. It provides a single point of global access to a class, just the way the singleton pattern does [Cooper]. The difference is that, for each new parameter, another instance is created.

The instances are stored in the cache. There is no mechanism to release the instances and allow them to be garbage collected, at present. In addition, just like any global store, if users alter the values of the instances incorrectly, the bug might be hard to trace.

Another drawback of the parametric singleton pattern is that there is no control over who accesses the instances. Just like the singleton, the parametric singleton is the only class that can create an instance of itself [Stelting].

An oft-cite drawback of the singleton pattern, which can be equally true of the parametric singleton pattern, is that when copies of the Singleton run in multiple VMs they will create an instance for each machine. Our implementation of the parametric singleton pattern will not suffer from this malady because the *locateRegistry* invocation will locate the registry no matter which virtual machine it is running on. However, this is a bug waiting to bite the unsuspecting parametric singleton (or non-parametric singleton) implementer [Fox].

The parametric singleton pattern leaves several topics for future work. The introduction of a check-in mechanism for inhomogeneous classes seems particularly useful. For example, in the case of a consumed resource (like a serial port) I may need to know who has the serial port and use some mechanism to release the serial port. A locked resource can cause a process to deadlock. Forcing the release of a resource will break the deadlock at the cost of causing unpredictable behavior.

We have found that already running RMI registries cause a similar problem. Breaking the deadlock means killing a running registry. The question of what happens when you kill the registry, remains open. Also open is exactly what would be a safe mechanism for killing a registry. Typically, different registries have different class paths. Restarting the registry not only enables a change in class path, but also a change in the execution context. A change in execution context requires a reloading of class files, enabling updates to code. A natural alternative is to create a new class loader, but this can be a very memory intensive approach, particularly since the number of class loaders is not bound. Worse, the use of multiple class loaders can defeat some implementations of the singleton design pattern [Geary].

Setting the code base on new registries may be important for some users. By default, the code base of the registry is the same as that of the parametric singleton pattern. In the future, a mechanism to change this might be welcome, for example:

```
// protocol + current directory (of the VM) + trailing slash
String codebase ="file:"+System.getProperty("user.dir")+"/";
System.setProperty( "java.rmi.server.codebase", codebase);
```

Some people claim that using lock files will prevent the launching of multiple singletons. We take exception to this claim, since applications that die before deleting the lock file will continue to cause deadlocks. Thus the question of how to resolve this issue remains open [JDF].

There has been some discussion on the RXTX group about unit testing the parametric singleton pattern. The question of how to implement this unit testing properly remains open. One (untested) idea is to create a façade design pattern in the form of a simplified interface and then use versions of this in order to provide a mock-up for the unit test [RXTX].

REFERENCES

[Beck97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1998.

[Cooper] James W. Cooper. *Java Design Patterns*. Addison-Wesley, 2000.

[Fox] Joshua Fox. "When is a Singleton not a Singleton?", http://java.sun.com/developer/technicalArticles/Programming/singletons/, January 2001. Last accessed April 14, 2006.

[Gamma et Al.] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.*Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Geary] David Geary. "Simply Singleton". Java World, April 25, 2003. http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns_p.html Last accessed April 15, 2006.

[Goldfedder] Brandon Goldfedder. *The Joy of Patterns*. Addison-Wesley, 2002.

[Grand] Mark Grand. *Patterns in Java, Volume 1*. John Wiley & Sons, Inc. 1998.

[JDF] Java Developer Forum. "Java Forums - Prevent launching multiple instances of a Java Application?" http://forum.java.sun.com/thread.jspa?threadID=565828&messageID=4101278 Last accessed April 15, 2006.

[Maclean] Stuart Maclean. "On The Singleton Software Design Pattern". Technical Report DSSE-TR-97-4, Dept of Electronics and Computer Science, University of Southampton, September 1997.

[RXTX] Douglas Lyon. "Sole of a new design pattern" in the RXTX forum, March 22, 2006. http://mailman.qbang.org/pipermail/rxtx/Week-of-Mon-20060320/006650.html Last accessed April 15, 2006.

[Shalloway] Alan Shalloway and James R. Trott. *Design Patterns Explained*. Addison-Wesley, 2002.

[Stelting] Stephen Stelting and Olav Maassen. *Applied Java Patterns*. Prentice Hall, 2002.

About the authors

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 30 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.

Francisco Catellanos earned his B.S. (Hons) degree in computer science at Western Connecticut State University. Francisco Castellanos worked at Pepsi Bottling Group in Somers, NY as a software developer. Currently he is working on a thesis to complete his M.S. degree in Electrical and Computer Engineering from Fairfield University. His research interests include grid computing. He is currently employed by Access Worldwide in Boca Raton, FL as a software developer. Email: fsophisco@yahoo.com