Pooling

Michael Kircher, Prashant Jain

{Michael.Kircher,Prashant.Jain}@mchp.siemens.de

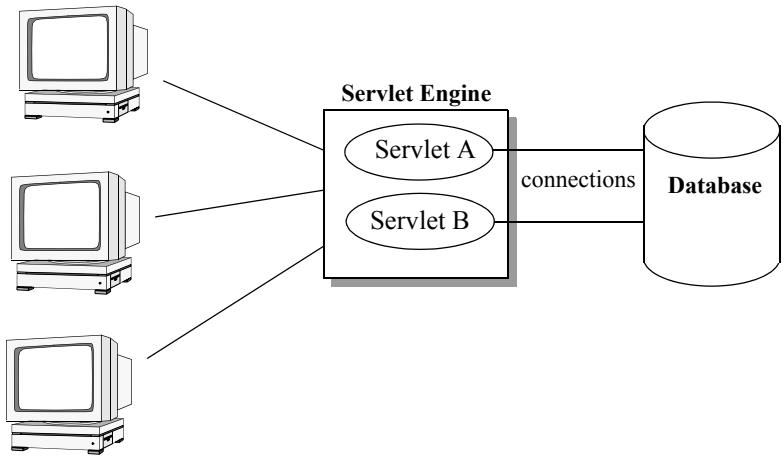
Corporate Technology, Siemens AG,

Munich, Germany

Pooling

The Pooling pattern describes how expensive acquisition and release of resources can be avoided by recycling the resources no longer needed.

Example Consider a web-based online catalog application that allows users to select and order one or more items. Assume that the solution is implemented in Java using a three-tier architecture. The clients consist of web browsers that communicate with a Java Servlet engine such as Tomcat. The business logic is implemented by one or more Java servlets that execute in the Java Servlet Engine. The servlets themselves connect to a database using a Java Database Connection (JDBC) interface. The following figure shows such a setup with two servlets executing in the Servlet Engine and connecting to a database.



Most of the web pages of the catalog are dynamically generated and depend upon the database for their contents. For example, to get a list of available items in the catalog along with their pricing, a servlet connects to the database and executes a query. The servlet uses the results of the query to generate the HTML page that is displayed to the user. Similarly, if a user makes a purchase and enters payment details via an HTML form, the servlet connects to the database and updates it.

In order to execute SQL statements on the database, a servlet needs to obtain a connection to the database. A trivial implementation would create a connection to the database on every request of the servlet. However, such a solution can be very inefficient since it incurs a delay in creating a connection to the database on every user request. In addition, such a solution is expensive since it results in potentially thousands of connections being created within a short time period.

An optimization to the solution could be to store a connection in the session context of the servlet engine. This would allow reuse of connection per session. Therefore, multiple requests from a user belonging to the same session would use the same connection to the database. However, an online catalog can potentially be accessed by hundreds of users simultaneously. Therefore, even with this solution, to meet the requirements , a large number of database connections would be needed.

Context Systems that continuously acquire and release resources of the same or similar type, and have to fulfill high scalability and efficiency demands.

Problem Many systems require fast and predictable access to resources. Such resources include network connections, instances of objects, threads, and memory. Besides providing fast and predictable access to resources, systems also require that the solution scales across the number of resources used, as well as the number of resource users. Also, different user requests should experience very little variation in their access time. Thus, the acquisition time for resource *r0* should not vary significantly from the acquisition time for resource *r1*, where r0 and r1 are resources of the same type.

To solve the above mentioned problems the following **forces** need to be resolved:

- *Performance*—Wastage of CPU cycles in repetitious acquisition and release of resources should be avoided.
- *Predictability*—The acquisition time of a resource by a user should be predictable even though direct acquisition of the resource from the resource environment can be unpredictable.
- *Simplicity*—The solution should be simple to minimize application complexity.
- *Stability*—Repetitious acquisition and release of resources can also increase the risk of system instability. For example, repetitious acquisition and release of memory can lead to memory fragmentation. The solution should minimize system instability.
- *Reuse*—Unused resources should get reused to avoid overhead of re-acquisition.

Patterns such as Eager Acquisition [Kirc02] and Lazy Acquisition [Kirc01] resolve only a sub-set of these forces. For example, Eager Acquisition allows for predictable resource acquisitions, whereas Lazy Acquisition focuses more on avoiding unnecessary acquisitions of the resources.

In summary, how can resource acquisition and access be made predictable while ensuring that system performance and stability are not compromised?

Solution Manage multiple instances of one type of resource in a pool. This pool of resources allows for reuse when resource users release resources they no longer need. The released resources are then put back into the pool.

To increase efficiency, the resource pool eagerly acquires a static number of resources after creation. If the demand exceeds the available resources in the pool, it lazily acquires more resources. To free unused resources several alternatives exist, such as those documented by the Evictor [Jain02] pattern or the Leasing [JaKi00] pattern.

When a resource is released and put back into the pool, it should be made anonymous by the resource user or the pool, depending on the strategy used. Later, before the resource is reused, it needs to be re-initialized. If a resource is an object, providing a separate initialization interface can be useful. The object ID, in many cases a pointer or a reference, is not used for identification by the resource user or the pool.

Structure The following participants form the structure of the Pooling pattern:

A *resource user* acquires and uses resources.

A *resource* is an entity such as memory or a thread.

A *resource pool* manages resources and returns them to resource users on acquisition requests.

A *resource environment,* such as an operating system, owns and manages resources initially.

The following CRC cards describe the responsibilities and collaborations of the participants

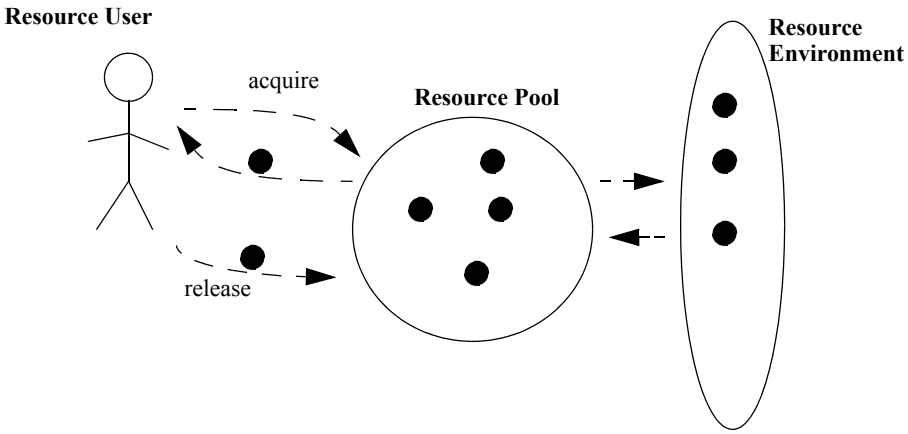
Class Resource User	***Collaborator***
Responsibility • Acquires and uses resources. • Releases unused resources to the resource pool.	• Resource

Class Resource	***Collaborator***
Responsibility • Represents a reusable entity, such as memory or a thread. • Is acquired from the resource environment by the pool and used by the resource user.	

Class Resource Pool	***Collaborator***
Responsibility • Acquires resources if necessary, eventually eagerly up front. • Recycles unused resources returned by resource users.	• Resource • Resource Environment

Class Resource Environment	***Collaborator***
Responsibility • Owns several resources initially.	• Resource

The interactions between the participants are shown in the following sketch.

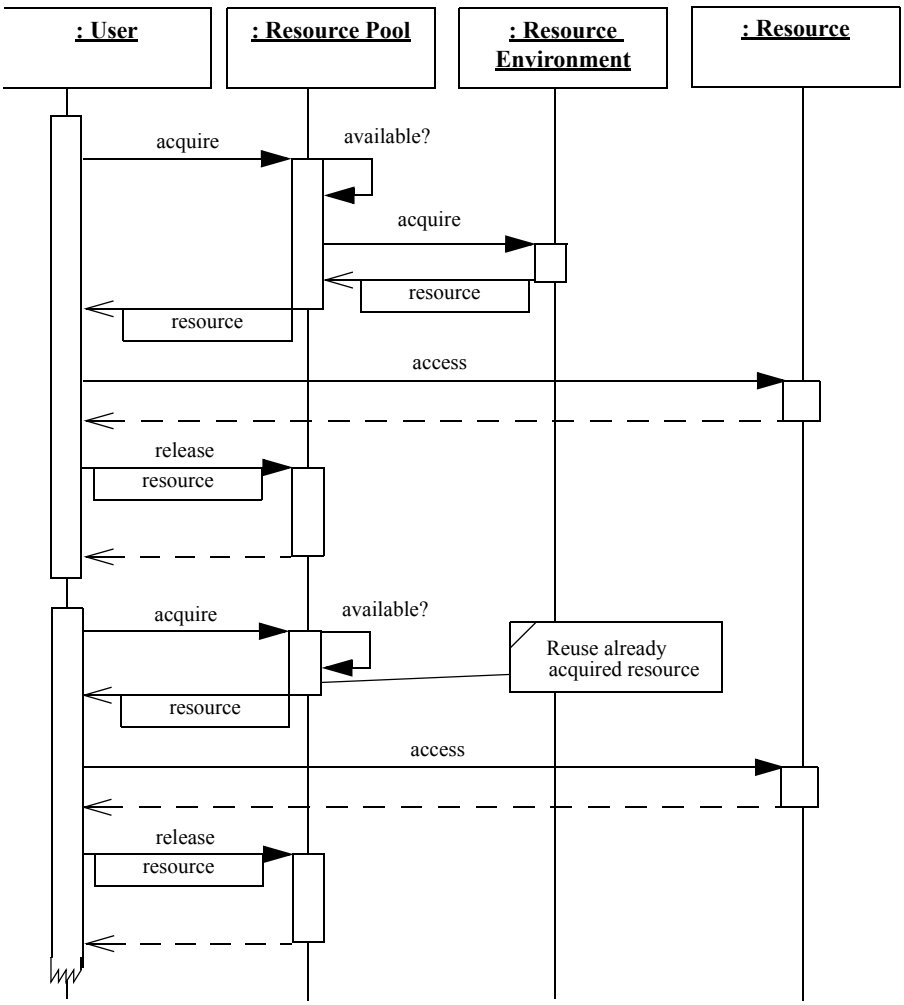


Dynamics The interaction between the participants varies slightly depending on whether the resource pool eagerly acquires resources at start-up or not.

Assuming the pool does acquire the resources up front, subsequent acquisition requests from resource users are served from those resources. Resource users release resources to the resource pool when no longer needed. The resources are then recycled by the pool and returned on new acquisition requests.

A slightly more complex situation is described in the following sequence diagram, where an acquisition request by a user leads to the acquisition of a resource from the resource environment. Acquisition on demand is done because no pre-acquired resources are available. The resource user accesses and uses the resource. When no longer needed, the user releases the resource back to the resource pool. The resource pool uses statistical data

about resource usage to determine when to evict the resources. The statistical data includes usage characteristics, such as last usage and frequency of usage.



Implementation There are eight steps involved in implementing the Pooling pattern.

1 *Select resources*: Identify resources that would benefit from being pooled in a resource pool. Simplify resource management by grouping resources by their types into pools. Grouping resources of different types into the same pool can complicate their management since it necessitates to provide multiple sub-pools and to perform ad-hoc lookups.

2 *Determine the maximum size of the pool:* In order to avoid resource exhaustion define the maximum number of resources that are maintained by the resource pool. The maximum number of resources available in the resouce pool equals the number of eagerly acquired resources plus the number of resources acquired on-demand [Kirc01]. The maximum number of resources is typically set at initialization time of the pool, but it is also possible to base it on configurable parameters such as the current system load.

3 *Determine the number of resources to acquire eagerly*: To minimize resource acquisition time at run-time, it is recommended to eagerly acquire at least the average number of resources typically used. This reduces acquisitions during the routine execution of the application. User requirements along with system analysis helps to

determine the average number of resources to be acquired. It is important to remember that too many eagerly acquired resources can be a liability and should be avoided.

4 *Define a resource interface*: Provide an interface that all pooled resources need to implement. The interface provides a common base class for all resource objects and facilitates in maintaining a collection of the resources. For example, an interface in Java may look like:

```
public interface Resource {}
```

An implementation of the `Resource` interface maintains context information that is used to determine whether to evict the resource as well as when to evict the resource. The context information includes timestamps and usage counts. For example, an implementation of the `Resource` interface in Java for a `Connection` class may look like:

```
public class Connection implements Resource
{
     public Connection () {
         // ....
     }
     // ....
     // Maintain context information to be used for eviction
      private Date lastUsage_;
      private boolean currentlyUsed_;
}
```

For legacy code integration where it may not be possible to have the resource implement the `Resource` interface, an Adapter [GHJV95] class can be introduced. The Adapter class can implement the `Resource` interface and then wrap the actual resource. The context information can then be maintained by the Adapter class. Here is what an Adapter class in Java for `Connection` class may look like:

```
public class ConnectionAdapter implements Resource
{
     public ConnectionAdapter (Connection connection) {
       connection_ = connection;
     }

     public Connection getConnection () { return connection_; }

     // Maintain context information to be used for eviction
     private Date lastUsage_;
     private boolean currentlyUsed_;
     private Connection connection_;
}
```

5 *Define a pool interface*: Provide an interface for acquisition and release of resources by resource users. For example, an interface in Java may look like:

```
public interface Pool
{
     Resource acquire ();
     void release (Resource resource);
}
```

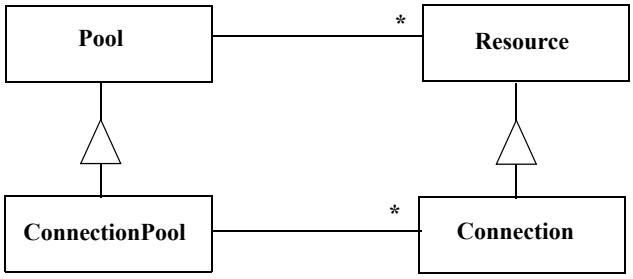
An implementation of the `Pool` interface would maintain a collection of `Resource` objects. When a user tries to acquire a resource, the resource would be fetched from this collection. Similarly, when the user releases the resource, it would be returned back to this collection. An example of an implementation of the `Pool` interface in Java that manages a collection of `Connection` objects may look like:

```
public class ConnectionPool implements Pool
{
```

```
        public Resource acquire () {
          Connection connection = findFreeConnection ();
          if (connection == null) {
              // eventually use a factory
              connection = new Connection ();
              connectionPool_.add (connection);
          }
          return connection;
        }

        public void release (Resource resource) {
             if (resource instanceof Connection)
              recycleOrEvictConnection ((Connection) resource);
        }

         // private helper methods

        private Connection findFreeConnection () {
             // ...
        }

        private recycleOrEvictConnection (Connection connection) {
             // ...
        }

         // ...
         // Maintain a collection of Connection objects
        private java.util.Vector connectionPool_;
    }
```

The code above shows one way of doing creation, initialization and eviction of connections. Of course, it is also possible to make those operations flexible, e.g. by using Strategy and a Factory [GHJV95].

The following class diagram shows the structure of the above mentioned classes.



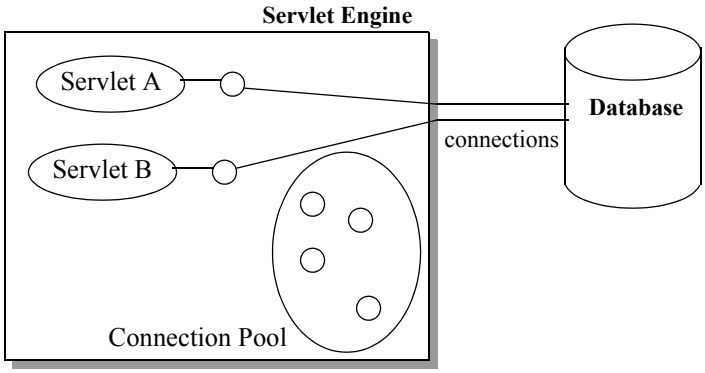
6 *Provide eviction of resources*: A large number of resources in the pool indicates that many unused resources exist wasting space and degrading performance. Therefore, to minimize system performance degradation and to shrink the pool to a reasonable size again, evict resources that have been used infrequently. An Evictor [Jain02] can be used for this purpose allowing different strategies to be configured to determine which resources to evict and how often to evict the resources. For example, the Evictor could use the attributes in the resource context to determine which resources to release.

7 *Determine resource recycling semantics*: Resource recycling varies depending upon the type of the resource. For example, recycling of a thread requires cleaning up of its stack and initialization of memory. In the case of objects, [Bish02] offers a way to decompose objects into multiple smaller objects, which are then recycled one-by-one.

8 *Determine failure strategies*: In case of any recoverable failure on resource acquisition or release, exceptions and/or error messages should be handled. If it is impossible to recover from a failure - exceptions should be thrown or a “null” resource, such as a

NULL pointer for malloc, should be handed back to the resource user. In case recycling fails, there are other patterns that can help minimize the impact of failure. For example, [Bish02] describes how to recylce broken objects by dividing complex objects up into smaller objects. These smaller objects are either recycled or re-created if broken and then reassembled again to the complex object.

Example Resolved In the case of the web-based shopping-cart example, the database connections are pooled in a connection pool. On every request by the user, the servlet acquires a connection from the connection pool, which it uses to access the database with queries and updates. After access, the connection is released to the connection pool in the context of the current request. As the number of concurrent requests is smaller than the number of current users, less connections are needed in total and expensive acquisition costs are avoided.



The figure above shows how individual servlets acquire connections from the connection pool to connect to the database. If the connection pool has no connection available and the maximum number of connections is not reached, new connections are created on demand.

Specializations The following specializations reflect applications of this pattern to specific domains, such as memory allocations, connection management, or object technology. Some of those specializations have been documented in existing literature.

Connection Pooling — Connection Pooling is regularly used in scalable frameworks/ applications which need to access remote services. Such remote services can be database services, e.g. via JDBC (Java Database Connectivity) or web servers, via HTTP.

Thread Pooling [PeSo97] — Thread Pooling is commonly used in environments that are inherently asynchronous. Highly scalable as well as real-time systems use Thread Pooling as their key mechanism to manage several threads. Using thread pooling allows such systems to avoid resource exhaustion by running too many threads. A typical thread pool starts with a limited number of eagerly acquired threads and lazily acquires more threads if it runs short on thread instances.

Component Instance Pooling [VSW2002] — Component Instance Pooling is typically used by application servers. Application servers optimize their resource usage by pre-instantiating a limited amount of commonly used components in order to serve the initial demand fast. When these components are all used up the application server instantiates new components as additional clients appear for them. If a client stops using a component, the component is put back into the pool.

Pooled Allocation [NoWe00] — Pre-allocate a pool of memory blocks, recycle them, when returned. Typically pooled allocators do not release the memory until system shutdown. Often multiple pools are installed, each with a different size of memory blocks. This avoids wastage of large memory blocks for requests of small size.

Object Pool [Gran98] — Manage the reuse of objects of a particular type that is expensive to create or only a limited number of that type can be created. An Object Pool is different from this pattern, because it assumes that every resource can be represented as an object, but this pattern does not make the restriction. The example and implementation sections of this pattern only use objects because it is easier to convey the key idea.

Variants The following variants describe related patterns, that are derived from this pattern by extending or changing the problem and the solution.

Mixed pool — Resources of different types are mixed in one pool. The interface of the pool might need to get extended in order allow resource users to differentiate their requests for resource acquisitions. All resources are managed by the same strategy. The interface has to allow for acquisition of resources of individual types.

Sub-pools — In some situations it is advisable to subdivide the pool of resources into multiple smaller pools. For example, in the case of a thread pool, you could partition the thread pool into multiple sub-pools. Each sub-pool could be associated with a certain priority (-range) [OMG02]. This ensures that thread acquisitions for low priority tasks do not cause resource exhaustion when high priority tasks need to be executed.

Known Uses. *JDBC Connection Pooling* [BEA02] — Java Database Connectivity (JDBC) connections are managed using Connection Pooling.

EJB Application Servers [IONA02][IBM02] — Application servers, that are based on component technology use Component Instance Pooling to efficiently manage the number of component instances.

Web Servers [Apac02] — Web servers have to serve hundreds, if not thousands, of concurrent requests. Most of them are served quickly, so creation of new threads per request is inefficient. Therefore, most web servers use Thread Pooling to efficiently manage the threads. Threads are reused after completing a request.

Car pooling — Car pooling is the sharing of multiple cars by a group of persons. Each person uses the car exclusively. This is typically done in big cities with a good public transportation system, where cars are only needed seldomly.

Consequences There are several **benefits** of using the Pooling pattern:

- *Performance*—The Pooling pattern can improve the performance of an application since it helps reduce the time spent in costly release and re-acquisition of resources. In cases where the resource has already been acquired by the resource pool, acquisition by the resource user becomes very fast.
- *Predictability*—The average number of resource acquisitions by the resource user executes in a deterministic time if those resources are acquired eagerly. With a proper eviction strategy in place, most acquisition requests can be served from the pool. As lookup and return of previously acquired resources is predictable, resource acquisition by the resource user becomes more predictable compared to always acquiring a resource from the resource environment.
- *Simplicity*—No additional memory management routines need to get invoked by the resource user. The resource user acquires and releases resources either transparently, using a Virtual Proxy [GHJV95], or directly from and to the resource pool.
- *Stability and Scalability*—New resources are created if demand exceeds available resources. Due to recycling, the resource pool delays resouce eviction based on the eviction strategy. This saves costly release and re-acquisition of resources, which increases overall system performance and stability.

- *Sharing*—Using the pool as a Mediator [GHJV95], unused resources can be shared between resource users. This also benefits memory usage and can result in an overall reduction in memory footprint of the application.
  Resources that have been acquired from the pool cannot be shared as long as they do not provide their own synchronization mechanism. This is because the pool does not synchronize access by any means.

There are also some **liabilities** using the Pooling pattern:

- *Overhead*—The management of resources in a pool causes a certain amount of CPU cycles. However, these CPU cycles are typically less than the CPU cycles required for release and re-acquisition.
- *Complexity*—Resource users have to explicitly release resources back to the pool of resources. Patterns such as Leasing [JaKi00] can be used to address this.
- *Synchronization*—In concurrent environments acquisition requests to the pool have to be synchronized to avoid race conditions and the possibility of corrupting the associated state.

See Also *Lazy Acquisition* [Kirc01] describes how to delay resource acquisition until the latest possible point in time in order not to waste unnecessarily. *Eager Acquisition* [Kirc02] describes how to eagerly acquire resources up front to allow for fast and predictable resource acquisition by the resource user. *Evictor* [Jain02] describes how to free unused resources by monitoring the resource usage. *Flyweight* [GHJV95] uses sharing to support large number of objects efficiently. The Flyweight objects can be maintained using Pooling.

This pattern is also known as *Resource Pool* [Bish02].

The *Caching* [KiJa02] pattern is related to *Pooling*, but *Caching* is about managing resources with identity. In the case of *Caching*, the resource user cares about which of the cached resources are returned. In the case of *Pooling* the resource user does not care about the identity of the resource since all resources in the pool are equal.

Acknowledgements Thanks to the patterns team at Siemens AG for their feedback and comments on earlier versions of this pattern. Special thanks to our EuroPLoP 2002 shepherd Uwe Zdun for his valuable feedback.

References

[Apac02] Apache Group, *Tomcat*, http://jakarta.apache.org, 2002

[BEA02] BEA, *Weblogic Server*, http://www.bea.com/products/weblogic/server, 2002

[Bish02] P. Bishop, *Recycle broken objects in resource pools*, Java World, http://www.javaworld.com, 2002

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Gran98] Mark Grand, *Patterns in Java - Object Pool*, Volume 1, John Wiley & Sons, 1998

[IBM02] IBM, *WebSphere Application Server*, 2002, http://www.ibm.com/websphere/

[IONA02] IONA, *iPortal Application Server*, 2002, http://www.iona.com/products/appserv.htm

[Jain02] P. Jain, *Evictor Pattern*, Pattern Language of Programs conference, Allerton Park, Illinois, USA, 2001, http://www.cs.wustl.edu/~pjain/papers/Evictor.pdf

[JaKi00] P. Jain and M. Kircher, *Leasing Pattern*, Pattern Language of Programs conference, Allerton Park, Ilinois, USA, August 13-16, 2000, http://www.cs.wustl.edu/~mk1/Leasing.pdf

[KiJa02] M. Kircher, and P. Jain, *Caching*, 2002, http://www.cs.wustl.edu/~mk1/Caching.pdf

[Kirc01] M. Kircher, *Lazy Acquisition Pattern*, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 5-8, 2001, http://www.cs.wustl.edu/~mk1/LazyAcquisition.pdf

[Kirc02] M. Kircher, *Eager Acquisition Pattern*, submitted to European Pattern Language of Programs conference, Kloster Irsee, Germany, July 4-7, 2002, http://www.cs.wustl.edu/~mk1/EagerAcquisition.pdf

[NoWe00] J. Noble, C. Weir, *Small Memory Software*, Addision-Wesley, 2000

[OMG02] Object Management Group (OMG), *Real-Time CORBA 1.0 Specification, Chapter 24.15 Thread pools,* http://cgi.omg.org/cgi-bin/doc?formal/01-12-35, 2002

[PeSo97] D. Petriu, G. Somadder, *A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers*, EuroPLoP 1997

[SSRBS00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture—Patterns for Concurrent and Distributed Objects,* John Wiley and Sons, 2000

[VSW2002] M. Voelter, A. Schmid, and E. Wolff, *Server Component Patterns - Component Infrastructures illustrated with EJB*, John Wiley & Sons, 2002