Resource Lifecycle Manager

Michael Kircher, Prashant Jain

Michael.Kircher@siemens.com

Corporate Technology, Siemens AG, Munich, Germany

pjain@gmx.net

IBM Research, Delhi, India

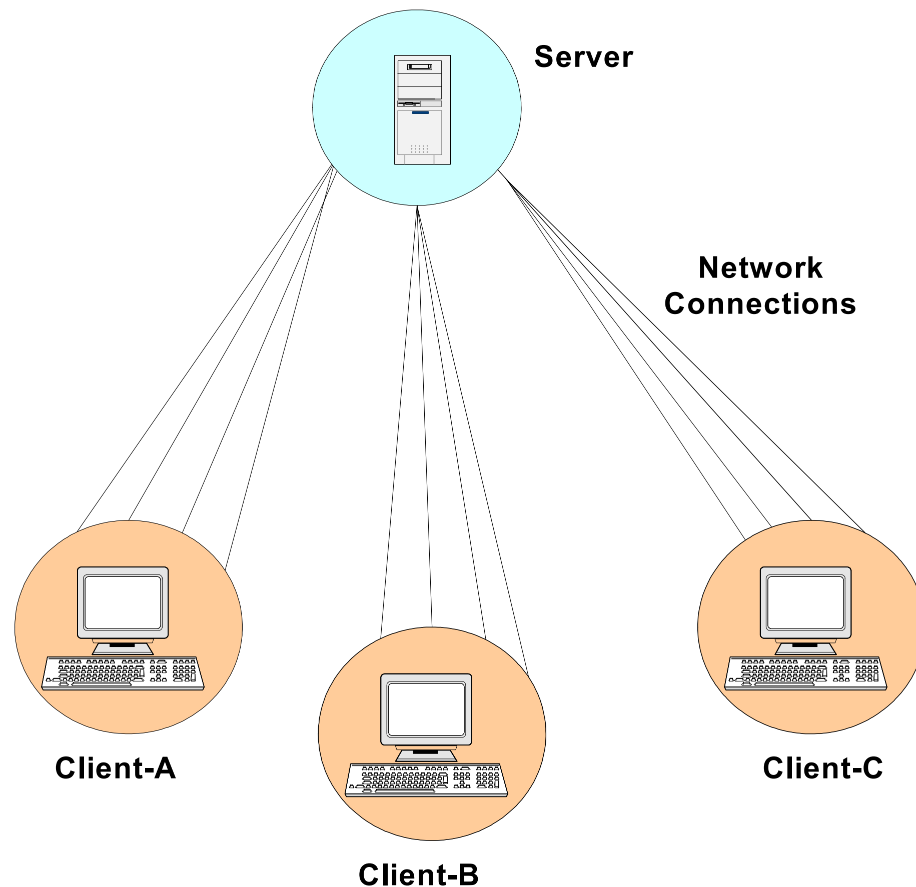
The Resource Lifecycle Manager pattern decouples the management of the lifecycle of resources from their use by introducing a separate Resource Lifecycle Manager, whose sole responsibility is to manage and maintain the resources of an application.

Note: This pattern appeared in a heavily reworked and updated version in the book Pattern-Oriented Software Architecture — Patterns for Resource Management published by Wiley [POSA3].

Example

Building distributed systems is challenging. Making distributed systems robust and scalable is even more challenging. The most important aspect of making distributed systems robust and scalable is how resources are managed. Resources in distributed systems can be of many different types, such as network connections, threads, synchronization primitives, servants etc. Network connections represent communication channels between client applications and distributed application services. Managing them efficiently requires the ability to determine when to establish connections and when to release them. Threads are especially important in distributed systems since they provide asynchronous behavior between different parts of an application, for example decoupling UI interaction from typical client functionality and service provisioning. However, managing threads effectively can be quite challenging since it involves close monitoring of their execution and the ability to determine when to create new threads or destroy no longer needed threads. Similarly, synchronization primitives such as locks and tokens are typically needed to synchronize the asynchronous parts of an application and to allow for their internal coordination and interaction. However, when and how these

synchronization primitives are created and released is very important and also very challenging to implement.



Consider a distributed system that needs to service thousands of clients. As a consequence, thousands of network connections are established between the clients and the server as shown in the figure above. The server typically provides one or more services to the clients. A client invokes a request on the server using its connection and as a result, the response to the synchronous invocation on the remote service is sent back via the same connection. If the request needs to be decoupled from the response and some form of asynchronous communication is required between the client and the server, then the server would need to open a new connection to the client and use it to initiate a call-back to the client. For example, the server may use a call-back to notify a client about events in the service.

Maintaining thousands of network connections becomes even more complex in the case of real-time systems where stringent QoS requirements exist and as a result, the server connection policies are typically quite complex. Connections might exist that are associated with specific priorities to be obeyed on executing requests on the service.

As connections depend on multiple low-level resources, they should be freed when no longer needed to ensure system stability. As a result, complex

lifecycle scenarios for each connection must be managed by the application. This complexity in managing the lifecycles of connections can affect the core functionality of the application and as a consequence can make the business logic of the application hard to understand and to maintain, because of the entangled connection management code.

Context

Resource users, such as applications, need to be decoupled from the management of the lifecycle of resources. Resources are acquired from a resource environment, such as an operating system.

Problem

The lifecycles of resources in small systems are typically controlled directly by them. But as the systems are further extended the resource users are faced with one or multiple of the following constraints:

- *Availability*—The number of available resources typically doesn't grow at the same rate as the size of the overall system. Therefore, in large systems, managing resources efficiently and effectively is important to ensure that they are available when needed by users.
- *Scalability*—As systems become large, the number of resources that need to be managed also grows and can become much more difficult to manage directly by the users.
- *Complexity*—Large systems typically have complex interdependencies between resources that can be very difficult to track. Maintaining and tracking these interdependence is important to allow proper and timely release of resources when they are no longer needed.
- *Performance*—To ensure that large systems don't face any performance bottlenecks, many optimizations are typically made. However, providing such optimizations can be quite complex if performed by individual resource users.

How can the above constraints be addressed thus allowing resource users to delegate the responsibility of managing the lifecycle of resources? In addition, how can the problem be solved while also addressing the following forces?

- *Stability*—If resource users have to care about resource lifecycle issues, they might forget to free resources, which leads in the long term to system instability. In addition, it should be possible to control acquisition of resources to ensure that there is no starvation of available resources at the system level leading to instability.

- *Inter-dependencies*—In complex systems resources of the same of different type might depend on each other. This means the lifecycle of resources are inter-dependent and need to be managed appropriately.
- *Flexibility*—The management of the resource lifecycle should be flexible by allowing support for different strategies. A strategy would provide a hook to allow configuration of how resource management should behave.
- *Transparency*—Resource lifecycle management should be transparent to the resource user. In particular, the resource user should not have to deal with any of the complexities of managing resources.

Solution

Separate resouce usage from resource management. Introduce a separate Resource Lifecycle Manager (RLM) whose sole responsibility is to manage and maintain the entirety of resources of an application. The RLM thus frees both the resources to be managed as well as their resource users from the task of proper resource management and thereby allows a system to provide high quality of service.

Users can use the RLM to retrieve and get access to specific resources. If a resource that is requested by a user does not yet exist, the RLM can also initiate its creation. In addition, RLM allows users to request an explicit creation of resources.

An RLM has knowledge of current resource usage and can therefore also reject a request for resource acquisition from a user. For example, if the system is running low on available memory, then the RLM can reject a user request to allocate memory.

The RLM also controls the disposal of the resources it manages, either transparently for the users or upon their explicit request. The RLM maintains its resources on the basis of appropriate policies that also take into account available computing resources like memory connections, and file handles.

If interdependencies between resources exist, the RLMs for individual resources have to work in “concert”. That means they have to maintain dependencies between resources, this can be done by one central RLM having a the full responsibility over individual, as well as dependent, resources, or by a separate RLM that only deals with the inter-dependencies while leaving the management of resources of the same type to resource-specific RLMs.

Structure

The following participants form the structure of the Resource Lifecycle Manager pattern:

A *resource user* acquires and uses resources.

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

A *resource lifecycle manager* manages the lifecycle of resources, including their creation/acquisition, reuse, and destruction.

A *resource environment*, such as an operating system, owns and manages resources. The resource environment might itself be a resource manager at the same or different abstraction level.

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

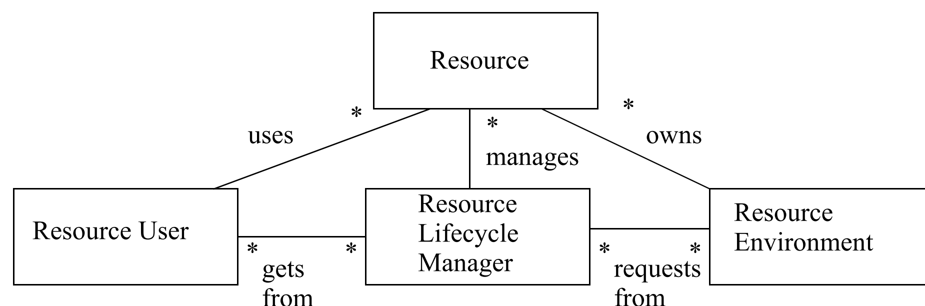
Class Resource User	***Collaborator***
Responsibility • Acquires and uses resources. • Releases unused resources to the resource lifecycle manager.	• Resource • Resource Lifecycle Manager

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

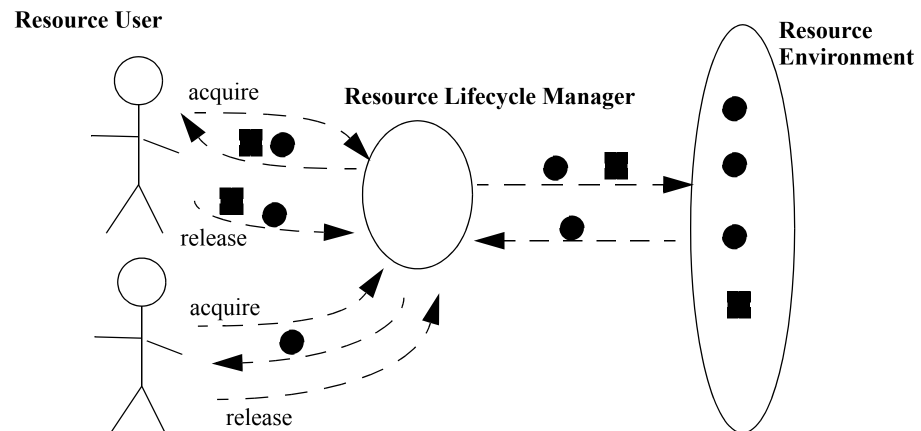
Class Resource Lifecycle Manager	***Collaborator***
Responsibility • Coordinates lifecycle of resources including creation/acquisition, reuse, and destruction.	• Resource • Resource Environment

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

The participants and their dependencies are displayed graphically in the following class diagram.



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

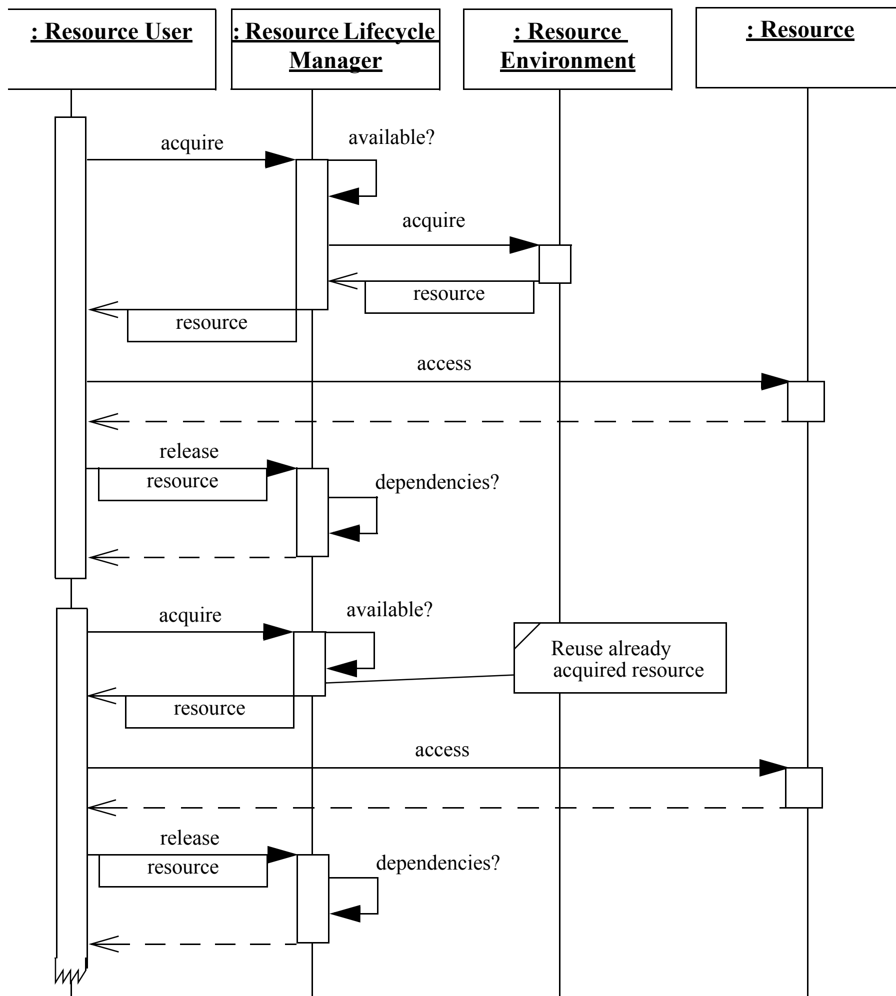


Dynamics

The dynamics of the pattern consists of the following activities:

- The system starts and initializes RLM.
- The resource user has a need for a resource and therefore tries to acquire the resource from the RLM.
- The RLM accepts the acquisition request and acquires the resource according to the resource acquisition strategy.
- The resource is handed to the user, which now uses it.
- The resource is accessed by the resource user.
- When the resource is no longer used by the user, it hands it back to the RLM.

- The RLM checks dependencies of the resource on other resources and decides to either recycle the resource or to evict it.



Implementation

There are seven steps involved in implementing the Resource Lifecycle Manager pattern.

1 *Determine resources that need to be managed*: A developer needs to first identify all resources whose lifecycle needs to be managed. Since resources can be of many different types, an application can provide multiple Resource Lifecycle Managers for different types of resources, for example, one Resource Lifecycle Manager for handling computing resources like processes, threads, file handles, and connections, and another Resource Lifecycle Manager for maintaining application components. On the other hand, a single Resource Lifecycle Manager can also handle resources of different types. Such a solution can be effective when complex interdependencies also need to be maintained among different types of resources [See Implementation Step 4]. If only

a single instance of Resource Lifecycle Manager is needed, it should be implemented as a Singleton [GHJV95].

2 *Define resource creation and acquisition semantics*: A developer needs to determine how resources will be created or acquired by the RLM. This includes determination of both *when* resources will be created/acquired as well as *how* resources will be created/acquired. Note that the RLM may create resources by combining more basic resources. Patterns such as Eager Acquisition [POSA3], Lazy Acquisition [POSA3], Allocation [Fern02], and Partial Acquisition [POSA3] can be used to control when resources will be acquired, while patterns such as Factory and Abstract Factory [GHJV95] can control how resources are created. Note, that resources are always acquired via the RLM. That is the only way it can control when, where, and how resources are acquired.
   In order to ensure system stability, the RLM might reject acquisition requests from resource users due to various reasons including the situation when available resources become scarce.
   Pooled resources are often created up-front during the initialization of the RLM using either Eager Acquisition or Partial Acquisition. Eager Acquisition creates/acquires a resource completely and before it is ever accessed—thus it is readily usable after its creation. However, it can take a long time to fully create/acquire large resources. Partial Acquisition can help reduce up-front acquisition time by performing step-wise resource acquisition. Finally, using Lazy Acquisition the entire creation/acquisition of a resource is deferred to the point in time it is actually accessed.

3 *Define resource management semantics*: One of the principal responsibilities of the RLM is to manage resources efficiently and effectively. Frequent acquisition and release of resources can be expensive and hence the RLM typically uses patterns such as Caching [POSA3] and Pooling [POSA3] to optimize the management of resources. Pooling can be used to keep a fixed number of resources constantly available. This strategy is in particular useful for managing critical computing resources like processes, threads, and connections, because they must be readily accessible. Caching, in contrast, keeps resources available in memory only for a certain amount of time. Caching is mostly applied for application components, because very likely they are used only for certain tasks. Once the tasks are performed, the components are not needed until the same tasks are executed again. Therefore, to not degrade the quality of service of an application, it can be helpful to remove unused components temporarily from memory, so that the space and the computing resources they occupy become available for components that are in use (see also the Passivation

pattern[VSW2002]). Therefore, using both these patterns together helps to keep a bound on total resource consumption.

4 *Handle resource dependencies*: In many applications resources are dependent on each other. So the first step is to isolate the different resource lifecycles in separate RLMs to ease maintenance. However, optimizations based on dependent resources will be hard to apply. In the example above, dependent resources can include application services that have been accessed via a specific connection. In real-time environments, servants implementing the application services are often directly associated with prioritized connections from clients so that priorities can be obeyed end-to-end. Therefore, the removal of any such connection influences the behavior of the servant, including its used resources. The servant might become inaccessible if dependent connections are evicted. Therefore, for managing connections and servants and their inter-dependencies, an RLM with a common responsibility should be considered. The exact applicability of such an RLM as well as its implementation is heavily dependent on application context as well as on the types of resources. Therefore, generic guidelines cannot be given on how to identify and handle resource dependencies.

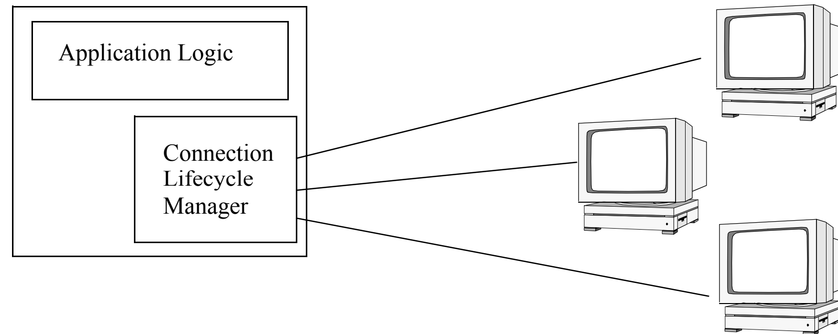
5 *Define resource release semantics*: Once resources are no longer needed, they should be automatically released by the RLM. Patterns such as Leasing [POSA3] and Evictor [POSA3] can be used to control when and how resources can be released. An Evictor allows for a controlled removal of less frequently used resources from the cache. To prevent the release of still referenced resources at all, an RLM can use the Leasing pattern that allows the RLM to specify the time for which the resources will be available to the users. Once this time expires, the resources can be safely and automatically be released by the RLM. Additionally, a Garbage Collector [JoLi96] can be used to identify unused resources, resources that are not referenced by any user or other resource, and evict them.

6 *Define resource access semantics*: Resources that have been created/ acquired need to be easily accessible. The RLM can use patterns such as Lookup [POSA3] to allow easy access to resources.

7 *Configure strategies*: For each of the above steps, the RLM should allow different strategies to be configured that control the final behavior of how the lifecycle of resources is managed. For example, if a resource is expensive, it should be acquired as late as possible using LA and released as early as possible using Evictor. On the other hand, a resource that is relatively less expensive and is used frequently should be acquired early on using EA and retained through the lifetime of the application. In dynamic environments reflection [POSA1] mechanisms can be employed to adapt configuration strategies according to the environment.

Example Resolved

Introduce a RLM that would assume the responsibility of connection lifecycle management and hence free the application from this responsibility. This, in effect, would decouple the management of connections from the business logic of the application.



The connection lifecycle manager would be introduced in the client and server parts of the distributed application. The clients would then use the connection lifecycle manager to request new connections to the server. Once these connections are given to the clients, their lifecycle is then managed by the connection lifecycle manager.

The responsibility of managing a connection is always assumed to be with the initiator of the connection, typically the client. However, there can be situations where, for example, a server may need to drop existing but idle connections in order to be able to accept new connections.

The eviction of unused connections can be triggered either by the application itself, or by the RLM implementation using the Evictor [POSA3] pattern.

Specializations

Object Manager - In building object-oriented systems, an Object Manager can be used as a specialized RLM that focuses exclusively on the management of objects.

Known Uses

Component Container—A container manages the lifecycle of application components and provisions application-independent services. Further, it manages the lifecycle of resources used by the components, see also Container and Managed Resource patterns in [VSW2002]. J2EE Enterprise Java Beans (EJB) [Sun03] and the CORBA Component Model (CCM) [OMG03b], are two concrete models that implement such functionality.

Remoting Middleware—Middleware technologies such as CORBA [OMG03a] and .NET Remoting [Ramm02] implement the RLM at multiple levels. Middleware ensures the proper lifecycle management of resources, such as connections, threads, synchronization primitives and servants implementing the remote services.

Current developments, such as Ice [ZeroC03], prove that CORBA and .NET are not the sole examples for middleware frameworks that implement RLM functionality. RLM is a proven concept in all middleware frameworks.

Modern distributed applications start off using such middleware and thereby rely on the RLM services provided by the middleware, freeing application logic from those issues.

Consequences

There are several benefits of using this pattern:

- *Efficiency*—The management of resources by individual users can be inefficient. The Resource Lifecycle Manager pattern allows coordinated and centralized lifecycle management of resources. This in turn allows for better application of optimizations and reduction in overall complexity.
- *Scalability*—Using the Resource Lifecycle Manager pattern allows for more efficient management of resources thus allowing applications to make better usage of the available resources, which in turn allows for higher application load.
- *Performance*—The Resource Lifecycle Manager pattern can ensure that various levels of optimizations are enabled to achieve maximum performance from the system. By analyzing resource usage and availability, it can use different strategies to optimize system performance.
- *Transparency*—The Resource Lifecycle Manager pattern makes it transparent to user how resources are managed. Different strategies can be configured to control resource creation/acquisition, management and release. By decoupling resource usage from resource management, RLM makes the life of a user easier.
- *Stability*—The Resource Lifecycle Manager pattern can ensure that a resource is allocated to a user only when sufficient amount is available. This can help make the system more stable by avoiding situations where user may directly acquire resources from the system causing resource starvation.

There are some liabilities of using this pattern:

- *Single point of failure*—A bug or error in the RLM can lead to the outage of large parts of the application. Redundancy concepts help only partly, as complexity is further increased and the performance is further throttled.
- *Flexibility*—When individual resource instances need a specialized treatment, the RLM pattern might be too inflexible.

See Also

Object Lifetime Manager [LGS99]—The Object Lifetime Manager is specialized on the management of singleton objects, as resources, in operating systems, that do not support static destructors properly, such as Real-Time operating systems.

Garbage Collector [JoLi96]—A garbage collector is specialized on evicting unused objects and their associated resources. Therefore, garbage collection is not a complete RLM as it does not deal with creation, allocation or acquisition.

Pooling [POSA3]—Pooling focuses specifically on recycling of resources, it does cover the complete lifecycle of resources

Caching [POSA3]—Caching is specialized on avoid expensive acquisitions and associated initialization of resources.

Manager [Somm98]—The Abstract Manager pattern focuses on the management of objects, not on general resource management.

Abstract Manager [Lieb01]—As the Manager pattern, the Abstract Manager pattern focuses only on the management of business objects in enterprise systems, not on general resource management.

Object Manager—The Object Manager pattern, as part of the Resource Management patter language by Frank Buschmann and Kevlin Henney, and this pattern base on the same ideas. Their work focuses on the compact Alexandrian form, we focus on the extensive POSA form.

Acknowledgements

Thanks to our EuroPLoP 2003 shepherd Ed Fernandez for his good comments and patience, when discussing the comments.

References

[Fern02] E. B. Fernandez, *A Pattern for the Request and Allocation of Limited Resources*, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 3-7, 2002, http://www.hillside.net/patterns/EuroPLoP/submissions-2002.html

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

[JoLi96] R. Jones, and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Son Ltd, 1996

[LGS99] D. Levine, G. Gill, and D. C. Schmidt, *Object Lifetime Manager*, Pattern Languages of Programming Conference, Allerton Park, Illinois, USA, 15-18 August,1999

[Lieb01] J. Liebenau, *Abstract Manager*, Pattern Language of Programs conference, Allerton Park, Illinois, USA, 2001

[OMG03a] Object Management Group (OMG), *Common Object Request Broker Architecture (CORBA)*, http://www.omg.org/cgi-bin/doc?formal/02-12-06, 2003

[OMG03b] Object Management Group (OMG), *CORBA Component Model Specification,* http://www.omg.org/cgi-bin/doc?formal/02-06-65, 2003

[POSA3] M. Kircher and P. Jain, *Pattern-Oriented Software Architecture — Patterns for Resource Management*, John Wiley and Sons, 2004

[Ramm02] I. Rammer, *Advanced .NET Remoting*, APress, 2002

[Somm98] Peter Sommerlad, *Pattern Languages of Program Design 3 — Manager pattern*, Addison- Wesley, 1998

[Sun03] Sun Micosystems Inc., *Java2 Enterprise Edition (J2EE) - Enterprise Java Beans (EJB)*, http://java.sun.com/j2ee/, 2003

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

[ZeroC03] ZeroC Inc., *Internet Communication Engine (ICE)*, http://www.zeroc.com, 2003