

A Pattern Language for Resource Management in Three Tier Architectures

Prashant Jain

Michael Kircher

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

Siemens AG, Corporate Technology

Munich, Germany

A Pattern Language for Resource Management in Three Tier Architectures

Highly robust and scalable three-tier systems must manage resources efficiently. A resource can be of many types including local as well as distributed services, database sessions and security tokens.

Resources are used in every tier. For example, in the presentation tier resources can be used in the form of network connections or GUI objects, in the application tier resources can be used in the form of loaded components, and in the persistency tier resources can be used in the form of database connections.

Following the layered architecture [POSA1], the user of the application tier is the presentation tier and similarly the user of the persistency tier is the application tier. Furthermore, the application tier is a resource provider for the presentation tier and similarly the persistency tier is a resource provider for the application tier.

Over time, a user of a running system may ask a resource provider for one or more resources. Resource acquisition can itself be expensive and therefore, a way is needed to reduce the initial cost of acquiring the resources. If the systems were to acquire all resources up front, a lot of overhead would be incurred and a lot of resources would be consumed unnecessarily. On the other hand, some systems may require highly deterministic behavior and therefore mandate acquisition of resources up front.

Once a resource has been acquired, the user can start using the resource. However, over time a user may no longer require some of the acquired resources. Unless the user explicitly terminates its relationship with the provider and releases the resources, the unused resources would continue to be needlessly consumed. This in turn can have a degrading effect on performance of both the user and the provider, as swapping or other memory management activities might occur. In addition, it can also affect resource availability for other users.

A resource need not be directly acquired by a user. The application tier itself may acquire multiple resources over time. For example, in servicing user requests from the presentation-tier, an application tier may acquire file handles. However, if it keeps on acquiring resources without ever releasing them, it will lead to resource exhaustion along with spontaneous, hard-to-recover errors.

To avoid this problem, the application may immediately release resources after using them. But the application may need to use the same resources again, which would require re-acquisition of those resources. However, re-acquisition of resources can itself be expensive and inefficient and should therefore be avoided by keeping frequently used resources in memory.

To address the above requirements of resource management requires the resolution of the following forces:

- *Optimality*: Unnecessary resource acquisitions should be avoided, as the acquisitions themselves are potentially expensive. In addition, the system load caused by unused resources must be minimized.
- *Simplicity*: The management of resources for a user should be simple by making it optional for the user to explicitly release the resources that it no longer needs.
- *Availability*: Resources not used by a user, or no longer available should be freed as soon as possible to make them available to new users. For example, resources associated with a network connection should be released once the connection is broken.
- *Lifecycle*: The frequency of use of a resource should influence the lifecycle of a resource.
- *Control*: Resource release should be determined by parameters such as type of resource, available memory and CPU load.
- *Actuality*: A user should not use an obsolete version of a resource when a new version becomes available.
- *Transparency*: The solution should be transparent to the user. The solution should incur minimum execution overhead and software development complexity.

To address these forces, the solution that is typically implemented by three-tier systems can be described using a set of design patterns, that is a pattern language. The pattern language abstracts away from any specific language, platform, or domain. Each pattern that is part of the pattern language can stand by itself and in fact has several known uses. However, in the context of three-tier systems, these patterns come together to form a pattern language that describes resource management in three-tier systems in a very succinct and precise manner.

The following are the primary patterns that address the above forces:

Lazy Acquisition [Kirc01]- describes how services including resources can be acquired on demand at the latest possible point in time in order to avoid unnecessary resource consumption.

Eager Acquisition - describes how services including resources can be acquired in advance to ensure their availability when they are needed.

Leasing [JaKi00]- describes the availability of a service in terms of time-based resource reservation with a service provider. This allows the service to cope with partial failure and to avoid accumulation of outdated and unwanted information.

Evictor [Jain01]- describes how resource consumption can be optimized by strategizing eviction of resources consumed by services.

References

- [GHJV] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
- [Jain01] P. Jain, *Evictor Pattern*, Pattern Language of Programs conference, Allerton Park, Illinois, USA, September 11-15, 2001
- [JaKi00] P. Jain and M. Kircher, *Leasing Pattern*, Pattern Language of Programs conference, Allerton Park, Illinois, USA, August 13-16, 2000
- [Kirc01] M. Kircher, *Lazy Acquisition Pattern*, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 4-8, 2001
- [POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal: *Pattern-Oriented Software Architecture—A System of Patterns*, John Wiley and Sons, 1996
- [POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann: *Pattern-Oriented Software Architecture—Patterns for Concurrent and Distributed Objects*, John Wiley and Sons, 2000