Explicit Interface
and
Object Manager

**Two patterns from a pattern language
for distributed computing**

Frank Buschmann

Siemens AG, Corporate Technology
Software and Engineering

`Frank.Buschmann@mchp.siemens.de`

Kevlin Henney

Curbralan Ltd

`kevlin@curbralan.com`

*"I wish life was not so short," he thought,
"Languages take such a time,
and so do all the things one wants to know about."*

J.R.R. Tolkien, The Lost Road

In this paper, we present two patterns from a proposed pattern language for distributed computing. We have distilled this language from our experiences in building distributed systems as well as from the distribution patterns that other software architects, designers, and developers have contributed to the software community. Almost all of the patterns in this language—92 of them to date—are taken from existing publications. However, two patterns are new: *Explicit Interface* and *Object Manager*. Both patterns repesent missing links that helped us to integrate the other patterns correctly and more tightly.

The patterns for distribution infrastructure, application infrastructure, event handling, concurrency, and synchronization of this language were presented and workshopped at the last two EuroPLoP conferences.

Most patterns in our language are well known and already published in the pattern literature. The two new patterns were introduced as a result of finding that the fit between some of the existing patterns was rough. Introducing *Explicit Interface* and *Object Manager* led to a snug fit leading to a clearer flow in the language:

> The *Explicit Interface* pattern is the central pattern of the component-partitioning patterns of our pattern language. It splits a component into two distinct physical entities: interface and implementation. This separates component usage aspects from realization details. Clients depend only on the signature and functional contract that the component interface defines, but not on this component's internal design. It reduces a client's dependency on the component's location, synchronization mechanisms, and other operational and developmental details.

> The *Object Manager* design pattern is one of the core resource management patterns in our language. It separates object usage from object management to support an explicit and centralized but decoupled and efficient handling of resource-based objects.

Much of the discussion within the *Explicit Interface* pattern about general component design was assigned initially to the *Extension Interface* [POSA2] pattern—in accordance with its original description [POSA2]. However, this led to problems when integrating the *Proxy* [GoF95] [POSA1] and *Facade* [GoF95] patterns into our language, as the focus of *Extension Interface* is on providing hook points for future
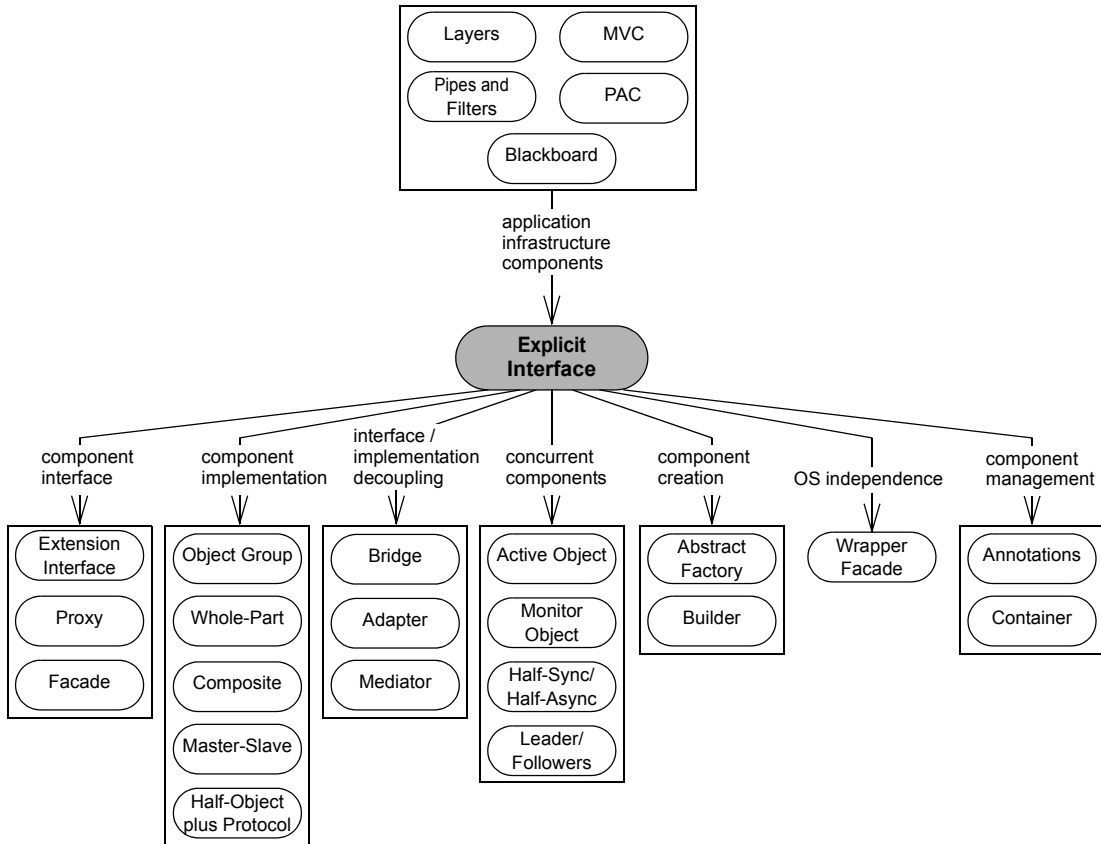
extensibility rather than on describing the separation between interface and implementation. Likewise, many of the statements about distributed and concurrent components that are now included in *Explicit Interface* were repeated across all of our application infrastructure patterns. This arrangement felt uncomfortable.

The resolution of our discomfort was simple: introduce a new pattern—*Explicit Interface*—that describes a fundamental partitioning principle for components in distributed systems. Hindsight reveals that this pattern is a cornerstone of any distributed architecture founded on a stable and explicit type system. Application infrastructure patterns can now reference *Explicit Interface* instead of repeating the same ideas. *Extension Interface*, *Proxy*, and *Facade* can focus on their essence: how to organize a component's interface.
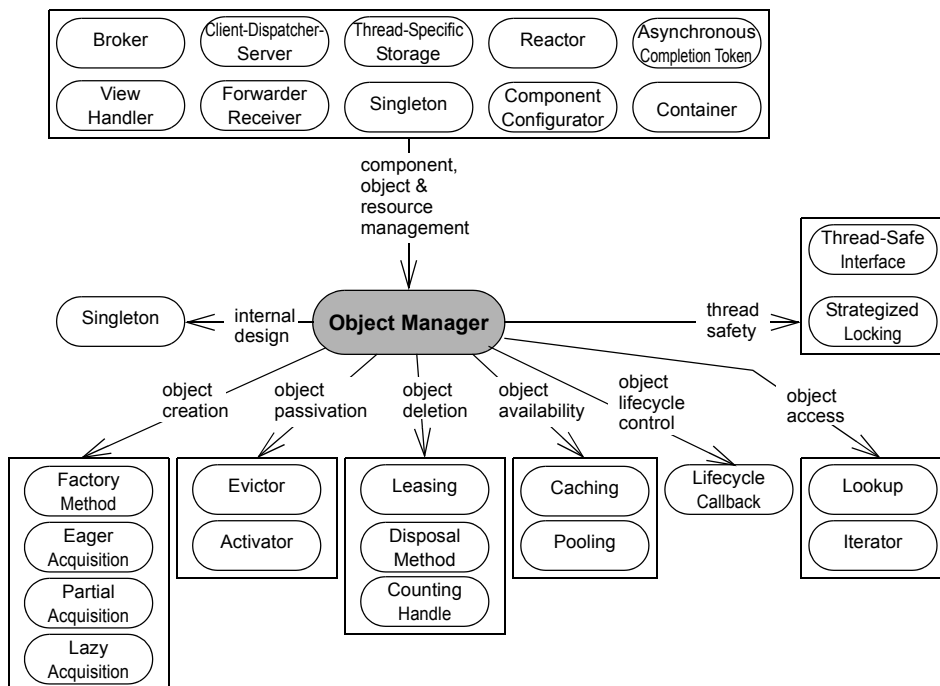
The name *Separated Interface* was used originally to describe what is now *Explicit Interface*. However, this name is already used to describe a related but subtly different pattern [Fow03]. To avoid confusion, and to allow us to make the link to this other pattern, the name *Explicit Interface* is now used.

The *Object Manager* pattern originates from the merger of two other patterns—*Manager* [Som97] and *Object Lifetime Manager* [LGS99]—and several discussions with our colleagues Michael Kircher and Prashant Jain, the authors of many of the resource management patterns included in our language. From these discussions we realized the need for a pattern like *Object Manager*, but in the body of software literature we could not find a single pattern that matched our purpose without too much stretching: *Manager* focuses on finding objects, mentioning object creation only briefly, and the scope of *Object Lifetime Manager* is limited to *Singletons* [GoF95] and static objects. Our solution was to fill this gap by introducing *Object Manager*, integrating the essence of *Manager* and *Object Lifetime Manager* into its description.
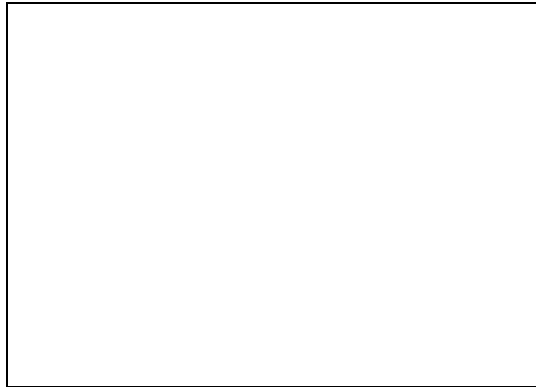
The following diagram shows how *Explicit Interface* is connected to the pattern language.

The next diagram show how the *Object Manager* pattern fits into our pattern language.

Explicit Interface **

We are designing components for a component-based system, or we are realizing the application functionality for a *Layers* [POSA1], *Pipes and Filters* [POSA1], *Blackboard* [POSA1], *Model-View-Controller* [POSA1], or *Presentation-Abstraction-Control* [POSA1] architecture.
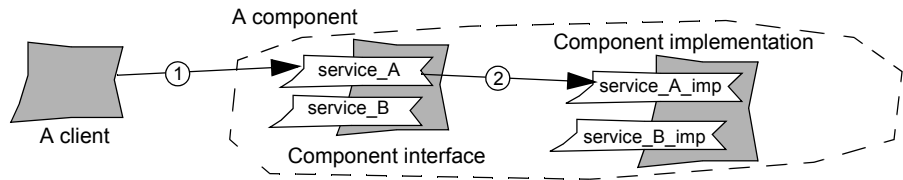
♦ ♦ ♦

A component represents an implementation of a self-contained unit of functionality and deployment with a published usage protocol. Clients can use a component as a building block in providing their own functionality. However, direct access to the full component implementation would lead to clients depending on the component internals, which ultimately increases an application's internal coupling.

Ideally, the only dependency of clients to a component should be on its published interface. If this interface remains stable, modifications in the implementation of the component should not affect its clients. Encapsulating components in 'ordinary' classes is therefore impractical: class interfaces are always bound to their implementations. An additional concern in distributed systems is location independence: clients of a component may reside in remote address spaces; in some cases, the location of the component may change during an applicaton's lifetime. Consequently, direct dependencies of a client to the particular location of a component should be avoided.

Explicit Interface **

Therefore:

Separate the interface of a component from its implementation so that the latter can be modified transparently and independently. Export the interface to the clients of the component, but keep its implementation and location private. A call from the client through this explicit interface will be to the component, but the client code will depend only on the interface and not on the component implementation.

♦ ♦ ♦

Splitting a component into two distinct physical entities—interface and implementation—separates component usage concerns from concrete realization and location details. The component interface is associated with a contract that clients must follow to use the an implementing component correctly [Mey97]. This contract includes operations offered by the component, the protocol for calling these operations, and any other constraints and information that clients must know to use the component correctly or most effectively. If clients do not satisfy their side of the contract, the component is entitled to fail. The quality of such failure is an additional design consideration. The component should depend on the contract and not vice-versa. The contract should not be added as an afterthought: the contract would inevitably depend on the component's internal implementation, they very thing that we are trying to insulate the client from. Any change to the component should require little more than a recompilation of the component and rebinding of the client

Extension Interfaces [POSA2], *Proxies* [GoF95] [POSA1], and *Facades* [GoF95] assume a proper separation of component interfaces from component implementation. They also help to provide a location-independent access to this implementation. The implementation's concrete design strongly depends on the component's concrete

purpose and responsibility. Domain-specific patterns and pattern languages can help with a proper component decomposition.
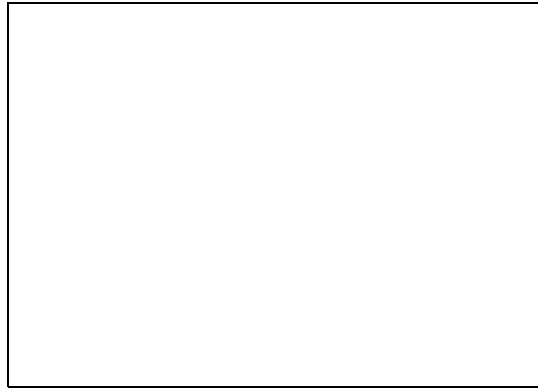
There are also infrastructure aspects to consider when partitioning a component. Typically, its entire implementation resides in a single physical location. This can incur penalties in a distributed system, however, from to latency, jitter, or remote method call failure. To optimize for quality of service it may therefore be beneficial to deploy the component across multiple network nodes. One option is replication: component instances are installed in multiple locations, for example, organized as an *Object Group* [Maf96]. Another option is to split component implementations into several distinct, yet cooperating parts—for example into *Whole-Part* [POSA1], *Composite* [GoF95], *Master-Slave* [POSA1], or *Half-Objects plus Protocol* [Mes95] arrangements—and to distribute these parts across the network. The quality of service of components can also benefit from concurrency: it allows them to handle multiple client requests simultaneously. *Active Objects* [POSA2] support the concurrency of large and complex components, *Monitor Objects* [POSA2] that of small, single class components. *Half-Sync/Half-Async* [POSA2] and *Leader/Followers* [POSA2] arrangements support the design of concurrent components that process network I/O. Using *Wrapper Facades* [POSA2] for accessing the operating system and platform-specific libraries helps to implement portable components.

Several options exist for expressing interfaces and implementations in terms of them. For example, Java and C# support the concept of an *Explicit Interface* in the core language, and classes can implement them directly. In other statically typed languages, such as C++, an *Explicit Interface* can be expressed as a fully abstract class. An implementing class is free to make other decoupling decisions, such as using a *Bridge* [GoF95], an *Adapter* [GoF95], or a *Mediator* [GoF95].

A component is often associated with an *Abstract Factory* [GoF95] or *Builder* [GoF95] that allows clients to obtain access to the component's interface and to transparently manage its lifetime. On platforms like the CORBA Component Model (CCM) [OMG99] and Enterprise JavaBeans (EJB) [MaHa99] components also come along with *Annotations* [VSW02]. *Annotations* specify how a particular component should be handled by a *Container* [VSW02].

Object Manager **

We must manage a set of objects or resources explicitly and with care. Or we are managing server resources in a *Broker* [POSA1] architecture or connections in a *Client-Dispatcher-Server* [POSA1] arrangement. Or we are handling objects in *Thread-Specific Storage* [POSA2], or event handlers in a *Reactor* [POSA2], or *Asynchronous Completion Tokens* [POSA2]. Or we are maintaining views in a *View Handler* [POSA1], or locations of remote peers in a *Forwarder-Receiver* [POSA1] configuration. Or we are managing components in a *Container* [VSW02]. Or we are controlling the lifetime of components managed by a *Component Configurator* [POSA2], or the lifetime of, and access to, *Singletons* [GoF95].

♦ ♦ ♦

Certain kinds of objects within an application—in particular server-side components, system resources, and singletons—require access control and a managed lifecycle. It is otherwise hard to maintain and use them efficiently, correctly, and without degrading the application's quality of service. However, implementing such functionality within the objects themselves overloads them with peripheral responsibilities and makes their simple and uniform use harder rather than simpler.

Likewise, clients should not be responsible for controlling the access to, and the lifecycle of, such objects: this would only make them dependent on the objects' internal structure, access constraints, and
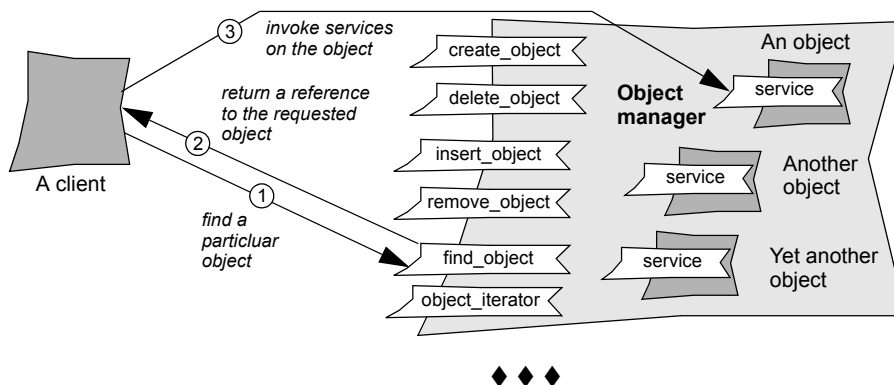
maintenance policies. Ultimately, such dependencies result in less encapsulated objects and in an increased coupling within the application. Ideally, therefore, a client should depend only on an object's usage interfaces and not its house-keeping obligations.

Different applications may also require different object management policies, which should therefore be configurable and changeable transparently for both the managed objects and their clients. In addition, the optimal management of a set of objects often depends on certain operational factors, such as the available memory. Object management must consider and embrace these factors.

Therefore:

Separate object usage from object lifecycle and access control. Introduce a separate object manager whose sole responsibility is to manage and maintain exclusively a given set of objects. Clients can use the object manager to gain access to objects with specific capabilities.

Where a client requests an object that does not yet exist, the object manager can create it on demand. Clients may also wish to request creation of objects explicitly via the object manager. In some situations the client may already have created the objects of interest, and may wish to hand custody of them over to an object manager. The manager should also control the disposal of the objects it manages, either transparently or in response to explicit client request. Let the object manager maintain 'its' objects on basis of appropriate policies that also take into account available computing resources such as memory, connections, and file handles.

♦ ♦ ♦

Object Manager **

An *Object Manager* frees both the managed objects and their clients from the task of detailed management. An *Object Manager* also introduces a localization–globalization trade-off: it concentrates and centralizes object management within an application while decentralizing and decoupling the management policy. Some applications provide only one *Object Manager* for a given type of managed object, sometimes—but not always wisely—implemented as a *Singleton* [GoF95]. Alternatively, an application can provide multiple *Object Managers* for different purposes and different contexts, for example, an *Object Manager* for handling threads and another for handling connections, or an *Object Manager* per thread to grant access to file resources. In general, there should be one *Object Manager* per group of objects that must be managed according to a specific set of policies. If an *Object Manger* is shared between multiple threads, it should offer a *Thread-Safe Interface* [POSA2].

Clients can request access to objects maintained by the *Object Manager* via its retrieval services. *Lookup* [POSA3] services allow a client to search for a specific object, for example, based on object names, object properties, or (index) keys. *Iterators* [GoF95] support manual traversal of multiple objects.

Internally, the *Object Manager* has several options to maintain the objects it manages. *Pooling* [POSA3] can be used to keep a fixed number of objects 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* [POSA3], in contrast, keeps objects available in memory only for a certain amount of time. *Caching* is mostly applied for application tasks. Once the application tasks have been performed, participating components are not needed until the same tasks are executed again. To avoid degrading an application's quality of service, it can therefore be helpful to drop unused objects and make the resources they occupy available for objects that are in use.

An *Evictor* [POSA3] allows for a *controlled* removal of less frequently used objects from the cache. Yet, objects that are evicted may still be referenced by clients. If the clients access the objects again, they must be re-activated. An *Activator* [Stal00] helps to provide the necessary object re-activation infrastructure.

To prevent the release of still-referenced objects, an *Object Manager* can choose from two other object removal policies. *Leasing* [POSA3] enables the *Object Manager* to specify the time that references to objects are valid, and to offer clients the opportunity to renew their leases. The *Object Manager* can destroy these objects safely once the lease has finally expired. *Counting Handles* [Hen01], in contrast, initiate the removal of an object as soon it is no longer referenced.

Objects maintained by an *Object Manager* must be created internally or passed in by clients. Registration functionality allows clients to hand over externally created objects into the custody of the *Object Manager* whereas *Factory Methods* [GoF95] support explicit object creation. Objects can also be created transparently for clients. Pooled resources are often created up-front during the initialization of the *Object Manager* using either *Eager Acquisition* [POSA3] or *Partial Acquisition* [POSA3]. *Eager Acquisition* creates an object 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 large objects. *Partial Acquisition*—which is also applicable to cached objects—reduces up-front creation time via step-wise object assembly. A third strategy is *Lazy Acquisition* [POSA3]: the complete creation of an object is deferred to the point it is first accessed. *Lazy Acquisition* is commonly applied with cached objects.

Objects maintained by an *Object Manager* must also be diposed of at some point in time. Deregistration functionality allows clients to take over the responsibility for objects from the *Object Manager*; *Disposal Methods* [Hen02b] to request the deletion of objects explicitly. When shutting the application (process) down, the *Object Manager* often disposes of all remaining maneged objects before it terminates, ensuring proper release of all resources used by these objects.

A set of *Lifecycle Callbacks* [VSW02] common to all objects allows the *Object Manager* to control their lifecycle uniformly, that is their initial creation, eviction, re-activation, and final disposal.

References

[Fow03]   M. Fowler: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003

[GoF95]   E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[LGS99]   D.L. Levine, C.D. Gill, D.C. Schmidt: *Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction*, C++ Report, vol. 12, Number 1, January 2000, http://www.cs.wustl.edu/~levine/research/ObjMan.ps.gz

[Hen01]   K. Henney: *C++ Patterns — Reference Accounting*, Proceedings of the 6th European conference on Pattern Languages of Programming, EuroPLoP 2001, Irsee, Universitätsverlag Konstanz, July 2002

[LGS99]   D.L. Levine, C.D. Gill, D.C. Schmidt: *Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction*, C++ Report, vol. 12, Number 1, January 2000, http://www.cs.wustl.edu/~levine/research/ObjMan.ps.gz

[Maf96]   S. Maffeis: *The Object Group Design Pattern*, Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies, USENIX, Toronto, Canada, June 1996

[MaHa99]   V. Matena, M Hapner: *Enterprise JavaBeans*, Version 1.1, Sun Microsystems, 1999

[Mes95]   G. Meszaros: *Half-Object plus Protocol*, in [PLoPD1], 1995

[Mey97]   B. Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, Englewood Cliffs, NJ, 1997

[OMG99]   Object Management Group: *CORBA Components Final Submission*, OMG TC Document orbos/99-02-05, February 1999

[PLoPD1]   J.O. Coplien, D.C. Schmidt (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995 (a book publishing the reviewed Proceedings of the First International Conference on Pattern Languages of Programming, Monticello, Illinois, 1994)

[PLoPD3]   R.C. Martin, D. Riehle, F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997 (a book publishing selected papers from the

Third International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1996, the First European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1996, and the Telecommunication Pattern Workshop at OOPSLA '96, San Jose, California, USA, 1996)

[POSA1]   F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture—A System of Patterns,* John Wiley and Sons, 1996

[POSA2]   D.C Schmidt, M. Stal, H. Rohnert, F. Buschmann: *Pattern-Oriented Software Architecture—Patterns for Concurrent and Networked Objects,* John Wiley and Sons, 2000

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

[Som97]   P. Sommerlad: *Manager*, in [PLoPD3], 1997

[Stal00]   M. Stal: *The Activator Design Pattern*, `http://www.posa.uci.edu/`, 2000

[VSW02]   M. Völter, A. Schmid, E. Wolff: *Server Component Patterns — Component Infrastructures illustrated with EJB*, John Wiley and Sons, 2002