Lazy Acquisition

Michael Kircher

Michael.Kircher@mchp.siemens.de

Siemens AG,

Munich, Germany

Lazy Acquisition

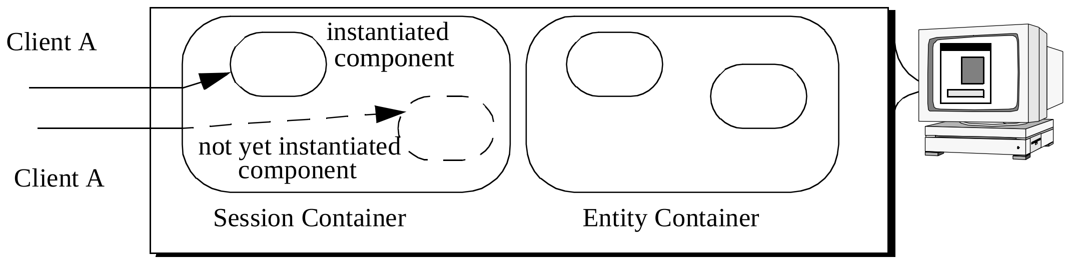
The Lazy Acquisition pattern defers resource acquisitions to the latest possible point in time during system execution in order to optimize resource usage. It is an abstraction of many existing patterns, that follow the same principle.

Example Assume your company is doing cutting-edge software development and therefore it always has a look at the latest technology specifications. One day your boss comes to you and asks you: “Michael, did you hear about this new CORBA Component Model technology.” You say: “Of course, I read the specification several times.” Saying this was a fault, you are now responsible for being one of the first to ever develop an application server conforming to the CORBA Component Model (CCM) specification. CORBA components[1] try to encompass many existing technologies, even aiming for compatibility with Enterprise JavaBeans. Components and containers are at the heart of the technology. The aim of component technology is on one hand to support better reuse, and on the other hand separation of concerns. The application programmer developing components, should not be bothered with things common to most components, such as transactions, security, persistency, etc.

So called component servers are hosting containers, which themselves host the components. Of course, a server would not be a real component server if it did not support many of these containers, and a container would be a poor container, if it supported only one component instance at a time. Scalability, is one of the key properties expected from a component server.

Applications based on component technology are normally structured as assemblies of components. The installation of an application on a component server involves typically many, possibly hundreds, of components. Loading every component of the application at its start-up would require many resources, such as memory, transaction IDs, or database connections, it should therefore be avoided.

How can you achieve scalability regarding the number of applications running at the same time in the same component server? One possible solution would be to extend your



hardware, e.g. the available memory or even setting up an additional machine, in order to provide more memory space for your components, but this can get very expensive, and, if the hardware is not yours, the person maintaining the hardware will definitely not be happy. To avoid any hardware changes, you could try to rely on OS mechanisms, such as virtual memory. As it is intended to be transparent, it does not provide full control and can heavily influence the stability of your system.

1. In the following, if I write about components, I refer to CORBA components, as they are defined by the CCM specification. CORBA components extend Enterprise JavaBeans to the world of CORBA, as it cannot provide “write once, run everywhere”, yet, the future has to prove its usefulness.

Context A system that must satisfy high demands, such as throughput and availability, while having restricted resources.

Problem Early acquisition of resources wastes resources, which can lead to high acquisition overhead and system instability.

Systems that have to manage many resources and/or expensive resource acquisition need a way to reduce the initial cost of acquiring the resources. If these systems were to acquire all resources up front, a lot of overhead would be incurred and a lot of resources would be consumed unnecessarily.

Each installed computing system (resource environment) has only a limited amount of resources, such as computing power, memory, storage, I/O interfaces, and other resources. Users should use those resources with care to avoid resource bottlenecks and contention. Resource bottlenecks are a common reason for system slow-downs and crashes. Efficient resource management is needed, even though it often complicates the application logic.

This leads to the following forces:

- *Contention* between resource users, which acquire them early and others which need them urgently, should be avoided. Component server providers are interested in running as many components as possible on their application servers while still providing high stability. How can they run many components on them and not risk instability?
- *Expensive acquisitions* can lead to unnecessarily high acquisition overhead. For example, if the component server mentioned above loaded and instantiated all components of all applications at startup time, it would take a long time to do so, thereby delaying its readiness for serving requests. Also, it would eventually consume more resources than available on the host, the resource environment, and would thereby trigger resource exhaustion and risk instability.
- *Transparency* to the user should be given in way that the solution should be transparent programming as well as complexity wise. E.g. the user of a component should not care about when the component is acquired and how that is accomplished. It only cares about a valid reference and proper service when accessing it.
- *Execution overhead* of the solution should be as little as possible. The component server should not introduce any noticeable delay or slow-down when executing requests on behalf of the user.

Solution Acquire resources at the latest possible point in time. The resource - such as a loaded shared library, the computing time for an evaluation of program expression, a new instance of a object, an initialized object, the state of an object - is not acquired until it becomes unavoidable to do so.

A Virtual Proxy [GHJV95] intercepts the use of a resource making it available dynamically - loading the shared library, evaluating the program expression, instantiating the object, initializing the object, or fetching the state of an object. Note that it is not mandatory to use a Virtual Proxy, much more it represents one typical solution to this problem.

The usage of these resources reduces the number of resources available by the resource environment, as acquisition actually takes the resource from the resource environment.

Structure The user access to the resource is intercepted by a Virtual Proxy. The Virtual Proxy transparently queries the resource environment for the resource. The resource environment hands back an instance of that resource to the Virtual Proxy. The Virtual Proxy now accesses the resource on behalf of the user.

Class User	***Collaborator***
Responsibility • Needs to access some information; unaware of the laziness.	• Resource

Class Resource	***Collaborator***
Responsibility • Provide the user requested functionality.	

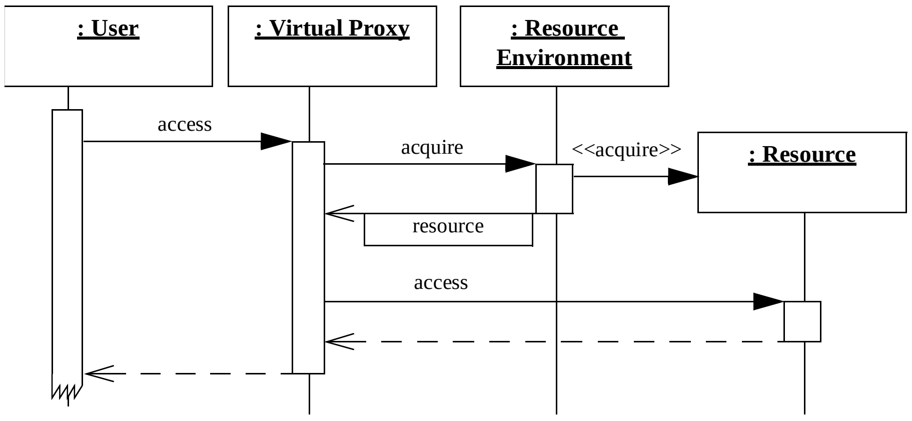
Class Virtual Proxy	***Collaborator***
Responsibility • Pretends to be the resource. • Provides the same interface as the resource. • Makes the actual resource available via the resource environment.	• Resource • Resource Environment

Class Resource Environment	***Collaborator***
Responsibility • Manages several resources. • Might recycle unused resources.	• Resource

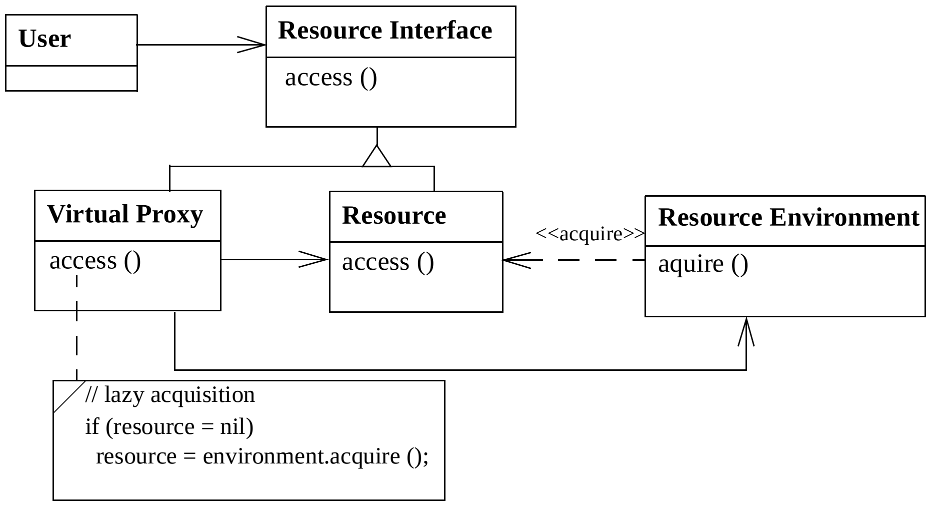
Dynamics The key dynamics of Lazy Acquisition is the interception of the first access of the resource by the user, which initially does not own the actual resource. At that time the resource is acquired. On the second and further access the usage continues normally, without any intervention.

As soon as a user accesses the resource, the Virtual Proxy intercepts this event and makes the resource available to them for use. The user does not notice the level of indirection by the Virtual Proxy.

As you may notice, this pattern describes how resources are consumed, but not how the resources are freed. Patterns that cover these aspects are for example Evictor [Jain02] [HeVi99] and Leasing [JaKi00].



The following UML diagram illustrates the structure of the Lazy Acquisition pattern.



It shows that the user only knows about the resource interface, it is transparent to the user if the Virtual Proxy or the Resource is accessed.

Implementation The implementation of this pattern is described by the following steps:

1 Define the interface by which the resource is accessed. This interface has to be provided by the actual resource as well as the Virtual Proxy.

2 Define the strategy by which the resource is actually obtained from the resource environment by the Virtual Proxy. The Strategy[GHJV95] pattern might be applied to switch on and off the laziness of acquisitions, e.g., by providing a regular and a lazy implementation of how resources are obtained from the resource environment.

3 Implement the Virtual Proxy and install the acquisition strategy in it. The Virtual Proxy hides Lazy Acquisitions on objects like resources. Use delegation to hide Lazy Acquisitions on functions.

Example Resolved Applying the lessons of system analysis the component server should not instantiate all components at startup. Instead, a scalable component server should defer instantiation of components to the time the component is actually needed. The need might arise from different situations, such as:

- A user wants to access a component of a specific class, but no component of that class is available.
- A new user wants to access a component of a specific class, but all pooled components of that class are busy serving users.

This principle is usually called lazy instantiation, which is the application of Lazy Acquisition to instantiation of objects and components.

To show how lazy instantiation works we give an example of how instantiation of a component can be delayed until the first actual access.

CCM component servers can be implemented at their heart by using standard CORBA 2.4 features, this means a CORBA component can then possibly consist of many CORBA objects, and the implementation of each is called a servant.

The following code segment shows the implementation of the `Container` as Virtual Proxy. The `Container` implements the `ServantLocator` interface which is part of the Portable Object Adapter (POA), the server-side dispatching mechanism of an Object Request Broker (ORB), at which all servants are typically registered with their Object ID.

A servant locator can be registered with a POA, instead of the actual servant, so that it gets pre-invoked and post-invoked on every dispatch to a servant by the POA. Moreover, pre-invoke allows late binding of the servant for dispatching. The servant locator can select which servant the operation is invoked on. In the implementation below we use this feature to find the servant via the object ID. If the servant has not been instantiated yet, we use the home to query for an instance.

```
class Container : public PortableServer::ServantLocator
{
public:
     // ...
     // ServantLocator interface
     virtual PortableServer::Servant preinvoke
                       (const PortableServer::ObjectId &oid,
                        ... )
     {
      PortableServer::Servant servant;
      if (servant_map_.find (oid, servant) == -1)
      {
          // We do not have a instantiated servant, yet.
          // Find the home for the object
          Home_var home = home_map_[oid];
          servant = home->get_servant_for_oid (oid);
          // Remember the servant
          servant_map_[oid] = servant;
      }
      return servant;
     }

     virtual void postinvoke ( ... );

     // Container specific method
     void register_home_reference
                       (const PortableServer::ObjectId &oid,
                        Components::KeylessCCMHome_ptr home)
     {
      home_map_[oid] =
          Components::KeylessCCMHome::_duplicate (home.in ());
     }
private:
     Servant_Map servant_map_;
     Home_Map home_map_;
};
```

In component architectures, such as CCM, so-called component homes are used to find existing component instances or to create them anew. The CCM specification dictates interfaces for homes, one of them is the `KeylessCCMHome`. It supports the `create_component` operation, which is used by clients to create new components - at least that is what the client believes it does. The home can transparently return only a valid object reference, without actually instantiating the component implementation, the servant.

```
class MyHome : public POA_Components::KeylessCCMHome {
public:
     // ...
     Components::CCMObject_ptr create_component ()
     {
      PortableServer::ObjectId oid = this->create_new_oid ();
      Components::CCMObject_var component_ref =
          contianer_poa_->create_reference_with_id (oid,
                                     "IDL:MyComponent:1.0");
      Components::KeylessCCMHome_var home = this->_this ();
      container_->register_home_reference (oid,
                                           home.in ());
      return component_ref;
     }
```

```
    PortableServer::Servant get_servant_for_oid
                      (const PortableServer::ObjectId &oid)
    {
     // Lazily Instantiate the servant for this object ID
     ComponentImplementation *component =
                              new ComponentImplementation;
     container_poa_->activate_object_with_id (oid,
                                              component);
     return component->_this ();
    }
private:
    Container *container_;
    PortableServer::POA_var container_poa_;
};
```

The container will use the `get_servant_for_oid` method, as soon as there is an actual request for the component. This can happen within the following microseconds, or hours later, depending on the usage pattern of the client.

The above code was only the component server framework code, no specialized code was shown. The actual component implements the following IDL:

```
interface Operations
{
    void operation_one ();
};
component MyComponent
    supports Operations;
```

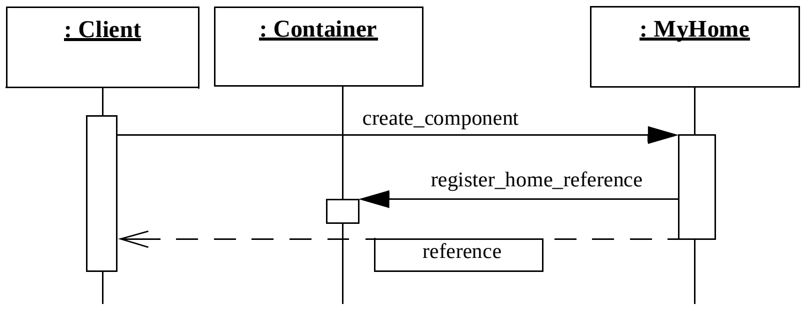
The component implementation does not see anything of the mechanisms internal to the container.

```
class ComponentImplementation : public POA_MyComponent
{
public:
    // ...
    void operation_one ()
    {
     // ...
    }
}
```

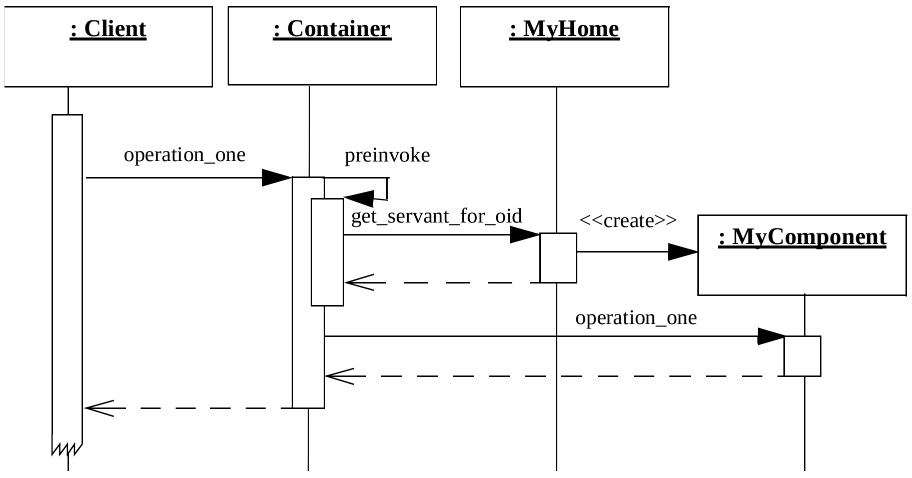
The next piece of code shows how the client requests a new instance of a component. The client then either uses the obtained object reference immediately to make a request on the component, or does some other stuff before invoking an operation for the first time on the component. In the latter case, especially if many components are created, a lot of initial instantiation overhead is saved by the component server.

```
// Use the object home as factory for new
// components
MyComponent_var component = myHome->create_component ();

// ... other things may happen

// Access the component by its interface;
// the actual servant is instantiated lazily
component->operation_one ();
```

Looking at all interactions we get the following sequence diagrams. The client first accesses the home to get a hold of a valid reference. At this point the actual component is not created, only a reference to it.



When the client actually accesses the component the first time., the actual component is created by the home.



Specializations Some specialized patterns derived from Lazy Acquisition are:

Lazy Instantiation - Defer the instantiation of objects/components until the instance is accessed by a user. As object instantiation is very often linked with dynamic memory allocation, and memory allocations are typically very expensive, this saves cost up front, especially for objects that are not accessed, but incurs a dramatic overhead in high-demand situations.

Lazy Loading - Defer the loading of a shared library until the program elements contained in that library are accessed. The Component Configurator Pattern [JaSc97] can be used to implement this. Lazy Loading is often combined with Lazy Instantiation, as for example objects need to be instantiated.

Lazy State [MoOh97] - Defer the initialization of the State [GHJV95] until the state is accessed. Lazy State is often used in situations where large state information is accessed rarely. This pattern becomes even more powerful in combination with Flyweight, or Memento [GHJV95].

Lazy Evaluation - Avoid evaluation of a node in a syntax tree if it its value is not of interest because of other nodes values. This approach can significantly improve performance of the evaluation, as unnecessary computations are avoided.

Lazy Initialization [Beck97] - Initialize the parts of your program the first time they are accessed. This pattern has the benefit of avoiding overhead in certain situations but has the liability of increasing the chance of accessing uninitialized parts of the program.

Variable Allocation [NoWe00] - Allocate and deallocate variable-sized objects as and when you need them. This specialization applies Lazy Acquisition specifically to memory allocations and deallocations.

Variants *Semi-Lazy Acquisition* - The idea is that you don't obtain the resource in the beginning but you also don't wait until the resource is actually needed. You load the resource some time in between. An example could be a network management system (NMS) where a topology tree of the network needs to be built. There are 3 options:

1) Build it when the application starts. Pro: The tree is available as soon as the application is initialized. Con: Slow start-up time.

2) Build it when the user requests it. Pro: Fast start-up time. Con: User has to wait for the tree to be constructed.

3) Build it after the application has started and before the user requests it. Pro: Fast start-up time and tree available when needed.

It is option 3 which is quite commonly used in NMS.

Known Uses Singletons [GHJV95], objects that exist uniquely in a system, are usually instantiated using lazy instantiation. In some cases Singletons are accessed by several threads. To avoid race conditions between threads during instantiation the Double-Checked Locking [SSRB00] idiom can be used.

The Haskel language, like other functional programming languages, allows lazy evaluation of expressions. Lazy evaluation means that an expression is not evaluated until the expression's result is needed to evaluate another expression. Lazy evaluation of parameters allows functions to be partially evaluated, resulting in higher-order functions which can then be applied to the remaining parameters. Evaluation of sub-conditions in a boolean expression in programming languages like Java or C++ is done using lazy evaluation in the form of short-circuiting operators e.g. &&.

EJB and COM+ application servers are containers that can host many different components at the same time. In order to avoid resource exhaustion they need to ensure that only components, which are actually by clients are active, others should be inactive. A typical solution is the application of the lazy loading and lazy instantiation patterns to the components running inside them. This saves valuable resources and assures scalability.

In ad hoc networking only temporal relationships between devices and their components exist, so that it gets too expensive to hold on to resources not actually needed at the current moment. This means that components need to be loaded, instantiated, destructed, and unloaded regularly. Therefore ad hoc networking frameworks need to offer mechanisms such as lazy loading and lazy instantiation. It is also possible to run lazy discovery of devices - the application will not be notified until the discovered device list changes from the last discovery run by the underlying framework [IrDA99].

A common feature of operating systems is to defer the complete loading of application libraries to the point in time when they are actually needed.

Just-in-Time (JIT) activation is an automatic service provided by COM+ that can help you use server resources more efficiently, particularly when scaling up your application to do high-volume transactions. When a component is configured as being JIT activated, COM+ will at times deactivate an instance of it while a client still holds an active reference to the object. The next time the client calls a method on the object, which it still believes to be active, COM+ will reactivate the object transparently to the client, just in time.

JIT compilation is heavily used in todays Java virtual machines (JVM). The compilation of the regular Java byte code into fast machine specific assembler code is done just-in-time. One of the virtual machines supporting this feature is the IBM J9 JVM. The opposite of JIT compilation is ahead-of-time (AOT) compilation.

JVM implementations optimize Java^{TM} class loading typically by loading the classes when the code of that class to be executed the first time. This behavior is clearly following the Lazy Loading pattern.

A Lazy Person is a person who avoids work. He/she waits until the latest possible point in time to do the job. For example a lazy student might not do his/her homework until just a few minutes before the teacher checks it, thereby fulfilling the force of not letting others know that he/she is actually lazy.

Laziness can become foolishness, if the person does not do the job at all. E.g. if the student does not do his/her homework at all, he/she might get kicked out of class.

Just-in-Time manufacturing, as used in many industries and lead by the automobile industry, follows the same pattern. Parts of an assembly are manufactured as they are needed. This saves cost of fixed storage.

Consequences There are several **benefits** of using the Lazy Acquisition pattern:

Scalability: Resources are saved until they are actually needed, this allows more efficient resource usage as the number of resources required at a specific point in time is reduced. Note that Lazy Acquisition only describes the loading of resources and that a pattern like Leasing [JaKi00] or Evictor [Jain02][HeVi99] might be needed to ensure the destruction and unloading of objects and components.

Reliability: The stability of the applications is increased because resource exhaustion becomes less likely.

Load Adaptation: The pattern enforces that only as many resources are acquired as are actually demanded. It thereby adapts resource consumption to system load.

There are some **liabilities** of using the Lazy Acquisition pattern:

Execution Overhead: Lazy Acquisition might incur a significant execution overhead due to the additional level of indirection.

Space Overhead: The pattern incurs a slight space overhead as additional memory is required for proxies resulting from the indirection.

Delay: The execution of the Lazy Acquisitions can introduce a significant time delay to the regular program execution. Especially for real-time systems such behavior is not advisable.

Complexity: The introduction of the pattern might introduce additional complexity to the system, especially concerning the program logic.

Predictability: The behavior of a lazy acquisition system can become distinctly non-linear when certain thresholds are reached, e.g. virtual memory systems can end up thrashing. This disqualifies the pattern for usage in real-time systems.

See Also The following two patterns are related to Lazy Acquisition:

Eager Acquisition [Kirc02] - The Eager Acquisition pattern can be conceived as the opposite of Lazy Acquisition. Eager Acquisition describes the concept of acquiring resources up front to avoid acquisition overhead at first access of clients

Pooling [KiJa02] - As both Lazy Acquisition and Eager Acquisition can be sub-optimal in some use cases, Pooling combines both into one pattern to optimize resource usage.

Other existing patterns with the word *lazy* in them, but not directly related to Lazy Acquisition are:

Lazy Optimization [Auer96] - Tune performance once the program is running correctly and the design reflects your best understanding of how the program should be structured.

Lazy Leader [Cold98] - The team leader involves himself in development to heavily and therefor fulfills his management responsibility only when really urged to.

Acknowledgements Thanks to Prashant Jain, Egon Wuchner, Frank Buschmann, Markus Völter, for their valuable comments on earlier versions of this pattern. Special thanks to Kevlin Henney, my EuroPLoP shepherd, for his excellent comments.

References

[Auer96]	Ken Auer: *Lazy Optimization: Patterns for Efficient Smalltalk Programming*, PLOPD 2, Addison-Wesley, 1996
[Beck97]	Kent Beck: *Smalltalk Best Practices Patterns*, Prentice Hall, 1997
[Cold98]	Jens Coldewey: *Lazy Leader Pattern*, http://www.coldewey.com/publikationen/Management/LazyLeader.8.html, 1998
[IrDA99]	Microsoft: *IrDA Protocol Overview*, http://www.microsoft.com/hwdev/infrared/IrDAapps.htm, November, 1999
[Jain02]	P. Jain, *Evictor Pattern*, to be workshopped at Pattern Language of Programs conference, Allerton Park, Illinois, USA, 2002
[JaKi00]	Prashant Jain and Michael Kircher: *Leasing Pattern*, Pattern Language of Programs conference, Allerton Park, Ilinois, USA, August 13-16, 2000
[JaSc97]	Prashant Jain and Doug C. Schmidt: *Service Configuratior - A Pattern for Dynamic Configuration of Services*, C++ Report, SIGS, Vol. 9, No. 6, June, 1997
[KiJa02]	M. Kircher and P. Jain, *Pooling Pattern*, submitted to European Pattern Language of Programs conference, Kloster Irsee, Germany, July 4-7, 2002
[Kirc02]	M. Kircher, *Eager Acquisition Pattern*, submitted to European Pattern Language of Programs conference, Kloster Irsee, Germany, July 4-7, 2002
[GHJV95]	Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
[HeVi99]	Michi Henning and Steve Vinoski: *Advanced CORBA Programming with C++ - Evictor Pattern*, Addison-Wesley, 1999
[MoOh97]	Peter Molin, and Lennar Ohlsson: *The Points and Deviations Pattern Language of Fire Alarm Systems - Lazy State Pattern*, PLOPD 3, Addison-Wesley, 1997
[NoWe00]	J. Noble, C. Weir, *Small Memory Software - Variable Allocation Pattern*, Addision-Wesley, 2000
[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