Eager Acquisition

Michael Kircher

Michael.Kircher@mchp.siemens.de

Corporate Technology, Siemens AG,

Munich, Germany

Eager Acquisition

The Eager Acquisition pattern describes how run-time acquisition of resources can be made predictable and fast by eagerly acquiring and initializing resources before their actual usage.

Example Consider an embedded telecommunication application with soft real-time constraints, such as predictability and low latency in execution of operations. The application is required to be reliable and predictable with no downtime.

In most operating systems, operations such as dynamic memory allocation can be very expensive. The time it takes for memory allocations via operations like "malloc" depends on the implementation of the operations. If the telecommunication application is supposed to run on a non-real-time operating system, the time to execute malloc will vary even if the same size of memory is to be acquired. Running the application on a real-time operating system (RT-OS) [Lynux][Wind], the time to execute malloc will most probably be constant, but the delay imposed by the operation will still be unacceptable.

Additionally, since neither the embedded computer hardware nor the RT-OS, e.g., [Wind], might have any memory management functionality, such as memory compaction mechanisms, the memory allocations can easily cause memory fragmentation.

Context A system that must satisfy high predictability and performance in resource acquisition time.

Problem In systems with soft real-time constraints, such as predictability and performance, dynamic acquisition of resources, e.g., memory and threads, at run-time is expensive. In addition, dynamic acquisition of resources incurs an unpredictable time overhead, especially in non-RT-OSs. As a result, the systems may not be able to fulfil the constraints imposed on them.

To solve the problem the following forces must be resolved:

- *Performance*—Resource acquisition by resource users must be fast.
- *Predictability*—Resource acquisition by resource users must be predictable. It should take the same amount of time each time a resource is acquired.
- *Initialization overhead*—Resource initialization at run-time needs to be avoided.
- *Stability*—Resource exhaustion at run-time needs to be avoided.
- *Fairness*—The solution must be fair with respect to other resource users trying to acquire resources.
- *Transparency*—The solution must be transparent to the resource user.

Comment 1: Michael, for each of the bullets above, how about having a keyword like we have for other patterns?

Comment 2: Good point. Changed. Can we really resolve Fairness and Stability?

Solution Eagerly acquire a number of resources before their actual usage.

At a time before resource usage, best at start-up, the resources are acquired from the resource environment. The resources are then kept in an efficient container, e.g., a hash map. Requests for resource acquisition from resource users are intercepted by an interceptor [SSRB00]. The interceptor accesses the container and returns the requested resource.

The point of time at which the resources are acquired can be configured using different strategies. The strategies should take into account different factors, such as when the resources will be actually used, the number of resources, their dependencies, and how long it takes to acquire the resources. Options are to acquire at system-startup or at a dedicated, eventually calculated [see Variants section], point in time after system-startup. Regardless of what strategy is used, the goal is to ensure that the resources are acquired and available before they are actually used.

Structure The following participants form the structure of the Eager Acquisition pattern:

A *resource user* acquires and uses resources.

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

A *virtual proxy* intercepts resource acquisitions by the user and hands the eagerly acquired resources back in constant time.

A *resource environment* manages several resources.

The following CRC[1] cards describe the responsibilities and collaborations of the participants.

Class Resource User	***Collaborator***
Responsibility • Acquires and uses resources. • Estimates resource usage and informs the Virtual Proxy.	• Resource • Virtual Proxy

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

Class Virtual Proxy	***Collaborator***
Responsibility • Provides the same interface as the resource. • Intercepts resource acquisitions by the user and hands back eagerly acquired resources in constant time.	• Resource • Resource Environment

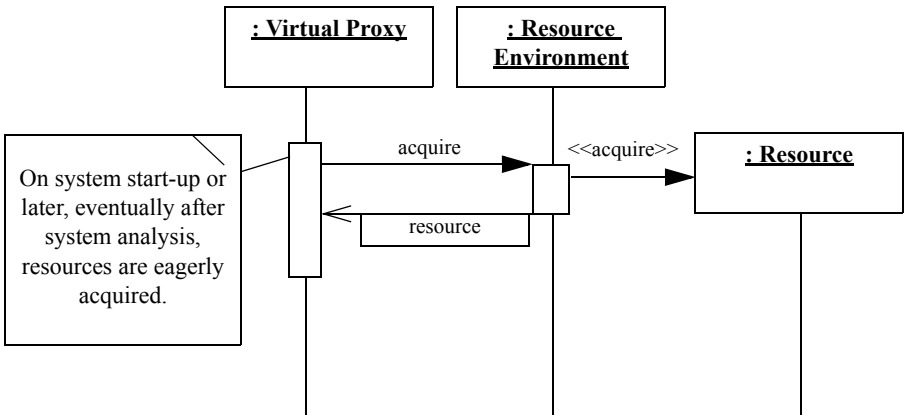
Class Resource Environment	***Collaborator***
Responsibility • Manages several resources. • Might recycle unused resources.	• Resource

The virtual proxy participant is responsible for the transparent integration of the Eager Acquisition pattern. It can be represented in an actual design in several ways including Virtual Proxy [GHJV95] and Interceptor [SSRB00].

Comment 3: Michael, instead of talking about "representing in actual design", why not say that "it can be implemented in several ways using Virtual Proxy and Interceptor"?

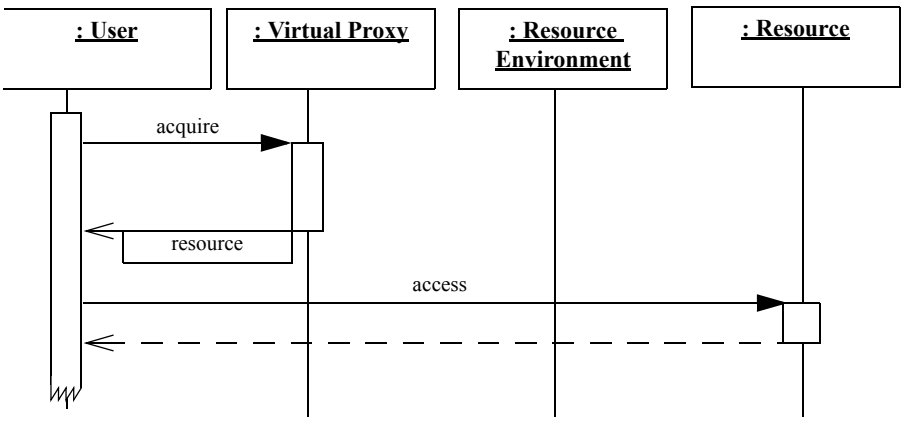
1. Class-Responsibility-Collaborators (CRC) cards help to identify and specify objects or the components of an application in an informal way, especially in the early phases of software development. A CRC-card describes a component, an object, or a class of objects. The card consists of three fields that describe the name of the component, its responsibilities, and the names of other collaborating components.

Dynamics The following sequence diagram shows how a resource is acquired up front and handed to the resource user on acquisition request.



Resources are acquired by the virtual proxy before their usage, at latest when the actual resource user is trying to acquire them. Resouce acquisition by the virtual proxy at system start-up time is advantageous in situations, where the resource usage can be fully predicted and if no time constraints are imposed on the system start-up time. Resource acquisition at system run-time is more flexible, but can cause overhead, of which the avoidance was one of the motivations for the pattern initially.

The resource user acquires resources, but get intercepted by the virtual proxy. The following sequence diagram shows how the resouce user acquires the resource:



The virtual proxy, obtained from e.g. a Factory [GHJV95] by the user, intercepts the acquisition request and returns an eagerly acquired resource. The resource user can now access and use the resource.

Implementation There are five steps involved in implementing the Eager Acquisition pattern.

1 *Select resources*: Determine the kind of resources to be eagerly acquired in order to guarantee predictable behavior of the overall system.
On investigation, check especially for the expensive acquisition of resources, such as

connections, memory, and threads. Those acquisitions are most likely to introduce unpredictability.

2 *Estimate resource usage*: Estimate the amount of resources a resource user acquires during its life-time. Perform test runs and measure the maximum resource usage, if it is not known up front.

3 *Integrate eager acquisition mechanism*: Introduce a level of indirection on the invocation path of the resource user to intercept resource acquisitions by the user.
Implement a container, such as a hash map, to hold the eagerly acquired resources. The container should allow predictable lookups, best is O(1) lookups.

4 *Determine timing strategy*: Decide on a strategy, when to eagerly acquire the resources:

At system start-up - Implement a hook so that the code for the eager acquisition is executed at start-up time. The advantage of acquiring resources at system start-up is that the run-time behavior of the system is not influenced.

Proactively during run-time - Use Reflection [POSA1] to detect system state that possibly leads to a need for resource acquisition by the resource user in the future. Proactive behavior has the advantage of being able to address acquisition requirements more closely, but this has to be paid by higher complexity and execution overhead for continuous system monitoring.

5 *Determine initialization semantics*: Decide on how to initialize acquired resources to avoid initialization overhead. For some resources, a complete initialization on pre-acquistion is impossible. In such cases the initialization overhead during run-time should be taken into account.
Depending on the kind of resource and its later usage, resources are anonymous, as long as they are not acquired by the resource user. Initialization, specific to a resource user, might be necessary after acquisition from the virtual proxy.

Comment 4: Michael, I found the last two sentences a bit confusing. Could you try to re-phrase?

Comment 5: Not even I understand what I wrote . I tried to fix it.

Example Resolved

Implement a custom `malloc()` function, which acquires up front memory blocks from the operating system, the resource environment. The acquired memory blocks are then handed back to the application code, the resource user, on subsequent function calls on `malloc()`.

Provide an extra `init()` function that is executed up-front to eagerly acquire memory blocks. Both functions, `init()` and `malloc()`, share a common memory area, where `malloc()` reads the eagerly acquired memory blocks from.

On some operating systems with poor memory compaction mechanisms, eagerly acquiring resources does not completely solve the problem. The application, respectively the newly implemented `malloc()` function, must avoid memory fragmentation. Memory fragmentation is influenced by the way the free-lists are categorized and managed.

Specializations

Eager Acquisition applied to specific domains results in:

Eager Instantiation - In this case objects are instantiated eagerly and managed in a list. When the application as resource user requests for new objects, new instances can be handed back from the list.

Eager Loading - Eager Loading applies eager acquisition to loading of libraries, such as class files in Java, shared objects on UNIX platforms, or dynamically linked libraries on Win32. The libraries are loaded up front - in contrast to Lazy Acquisition [Kirc01].

Variants The following list contains variants on the Eager Acquisition pattern:

Fixed Allocation, which is also known as Static Allocation [NoWe00], or Pre-Allocation, applies Eager Acquisition to allocation of memory. Fixed Allocation is especially useful in embedded, and real-time systems. In such systems memory fragmentation and predictability of the system behavior are more important than dynamic memory allocations.

Proactive Reallocation [CrLa01] means that resource acquisitions are made up front, but based on indications derived from resource usage by reflection techniques, and therefore not purely based on estimations.

Known Uses **Ahead-of-time compilation** - is commonly used by Java(TM) virtual machines to avoid compilation overhead during execution.

Pooling - Pooling solutions, such as connection or thread pools typcially pre-acquire a number of resources, such as network connections, or threads, in order to quickly serve first requests.

NodeB - In the software of the Siemens UMTS base station 'NodeB' the connections to various system parts are eagerly acquired at system-startup. This avoids unpredictable resource acquisition errors and delays during system run-time.

Hamster - A real-world known use is a hamster. It acquires as many fruits as possible before eating them in its cave. The hamster stores the food in its cheek pouch. Unfortunately, no numbers are available regarding its estimations about how much to acquire.

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

- *Predictability*—Availability of resources is predictable as requests for resource acquisitions from the user are intercepted and served instantly. Unnecessary variation in delay in resource acquisition, incurred by the OS, for example, are avoided.
- *Efficiency*—Since resources are already available when needed, they can be acquired within a constantly short time more efficiently.
- *Flexibility*—Customization of resource acquisition can easily be applied. Interception of resource acquisition from the user allows for strategized acquisition of resources by the virtual proxy. This is very helpful in order to avoid side effects, such as memory fragmentation.

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

- *Management Responsibility*—Management of eagerly acquired resources becomes an important aspect as not all resources might immediately get associated with a resource user and therefore need to be organized. Caching [KiJa02b] and Pooling [KiJa02a] patterns can be used to provide possible management solutions.
- *Static Configuration*—The system becomes more static, as the number of resources has to be estimated up front. Overly eager acquisitions must be avoided in order to guarantee fairness among resource users and to avoid resource exhaustion.
- *Over-acquisition* - Too many resources might be acquired up front by a sub-system, which might not need all of them. The resources are then missing from the resource environment, so that other sub-systems could use it. This can lead to unnecessary resource exhaustion. However, properly tuned resource acquisition strategies can help addressing this problem.

Slow system start-up—If many resources are acquired and initialized at system start-up, a possibly long delay, due to eager acquisitions, is incurred by the pattern. If resources are not eagerly acquired at system start-up, but later, there is still overhead associated with it.

See Also The opposite of Eager Acquisition is Lazy Acquisition [Kirc01], which allocates resources just in time - at the moment the resources are actually used.

The Pooling pattern [KiJa02a] combines the advantages of Eager and Lazy Acquisition into one pattern.

Acknowledgements Thanks to the patterns team at Siemens AG, to the EuroPLoP 2002 shepherd Alejandra Garrido, and the EuroPLoP 2002 writer's workshop for their valuable comments and feedback.

References

[CrLa01] Joseph K. Cross, and Patrick J. Lardieri, *Proactive and Reactive Resource Reallocation in DoD DRE Systems*, OOPSLA 2001 Workshop on "Towards Patterns and Pattern Languages for OO DRE Systems" (http://www.cs.wustl.edu/~mk1/RealTimePatterns), October, 2001

[IBM02] IBM, *Ahead-Of-Time Compilation*, J9 Java Virtual Machine, 2002

[Jain02] P. Jain, *Evictor Pattern*, Pattern Language of Programs conference, Allerton Park, Illinois, USA, 2002

[JaKi00] P. Jain and M. Kircher, *Leasing Pattern*, Pattern Language of Programs conference, Allerton Park, Ilinois, USA, August 13-16, 2000

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

[Kirc01] M. Kircher, *Lazy Acquisition Pattern*, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 5-8, 2001

[KiJa02a] M. Kircher and P. Jain, *Pooling Pattern*, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 4-7, 2002

[KiJa02b] M. Kircher, and P. Jain, *Caching*, 2002, http://www.cs.wustl.edu/~mk1/Caching.pdf

[Lynux] LYNUXWORKS, *LynxOS*, http://www.lynuxworks.com/products/lynxos/lynxos.php3, 2002

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

[Pryc02] N. Pryce, *Eager Compilation and Lazy Evaluation*, http://www.doc.ic.ac.uk/~np2/patterns, 2002

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture—Patterns for Concurrent and Distributed Objects,* John Wiley and Sons, 2000

[Wind] Windriver, *VxWorks*, http://www.windriver.com, 2002