Caching

Michael Kircher, Prashant Jain

Michael.Kircher@siemens.com

Corporate Technology, Siemens AG., Munich, Germany

pjain@gmx.net

IBM Research, Delhi, India

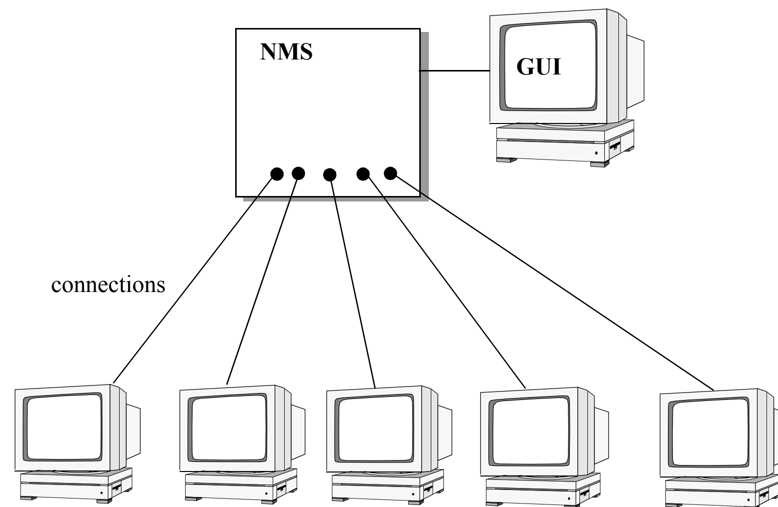
The Caching pattern describes how to avoid expensive reacquisition of resources by not releasing the resources immediately after their use. The resources retain their identity, are kept in some fast-access storage, and are re-used to avoid having to acquire them again.

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

Consider a network management system (NMS) that needs to monitor the state of many network elements (NEs). The NMS is typically implemented as a three-tier system. End-users interact with the system using the presentation layer -- typically a GUI. The middle-tier comprising the business logic interacts with the persistence layer and is also responsible for communicating with the physical NEs. Since a typical network consists of thousands of NEs, it is very expensive to set up persistent connections between the middle-tier, the application server, and all the NEs. On the other hand, an end-user can select any of the NEs using the GUI to get details of the NE. The NMS must be responsive to the user request and hence should

provide low latency between user selection of an NE and the visualization of its properties.



Establishing a new network connection for every network element selected by the user and destroying it after usage incurs overhead in the form of CPU cycles inside the application server. Also, the average delay to access a network element can be too slow.

Context

Systems, that repeatedly access the same set of resources and need to optimize for performance.

Problem

Repetitious acquisition, initialization, and release of the same resource causes unnecessary performance overhead.

In situations when the same component or multiple components of a system access the same resource, repetitious acquisition and initialization incurs cost in terms of CPU cycles and overall system performance.

Memory must be constantly allocated and later released to accommodate these resources. The cost of acquisition, access, and release of frequently used resources should be reduced to improve performance.

To address the problem the following forces need to be resolved:

- *Performance* — The cost of repetitious resource acquisition, initialization, and release must be minimized.
- *Usage Complexity* — The solution should not make acquisition and release of resources more complex and cumbersome.

- *Implementation Complexity* — The solution should not add unnecessary levels of indirection to access resources.

Solution

Temporarily store the resource in a (cheap) buffer called a cache. Subsequently, when the resource is to be accessed again, use the cache to fetch and return the resource instead of acquiring it again from the resource environment such as an operating system that is hosting resources. The Cache identifies resources by their identity, such as pointer, reference, or primary key.

To keep around frequently accessed resources and not release them helps avoid the cost of (re-)acquisition and release of resources. Using a cache eases management of components that access the resources.

When resources in a cache are no longer needed, they are released. The cache implementation determines how and when to evict resources no longer needed. Alternatively, this behavior can be controlled by strategies.

The Caching pattern is different from the Pooling pattern [POSA3] since Caching maintains the identity of a resource in memory; Pooling relies on a resource becoming anonymous and possibly gaining a different identity later.

Structure

The following CRC cards show how the participants interact with each other

Class Resource User	***Collaborator***
Responsibility • Acquires the resource from the Resource Environment. • Accesses the resource to retrieve data.	• Resource • Resource Environment

Class Resource	***Collaborator***
Responsibility • An entity, such as memory or a connection. • Is acquired from the resource environment either directly by the resource user or indirectly by the resource cache.	

Class Resource Cache	***Collaborator***
Responsibility • Buffers resources. • Eventually evicts resources.	• Resource • Resource Environment

Class Resource Environment	***Collaborator***
Responsibility • Manages several resources.	• Resource

The resource environment, e.g., an operating system, hosts the resources that are initially acquired by the resource user. The resource user then accesses the resource. When no longer needed, the resource is released to the cache. The resource user uses the Cache to acquire resources, that it needs to access again. Acquisition of a resource from the Cache is cheaper with respect to CPU utilization and latency compared to acquisition from the resource environment.

The explicit release to and re-acquisition from the Cache introduces usage complexity. This can be alleviated by using a Virtual Proxy [GHJV95] or Interceptor [POSA2] that makes the operations transparent. However, the level of indirection as a result of the lookup in the Cache cannot be avoided.

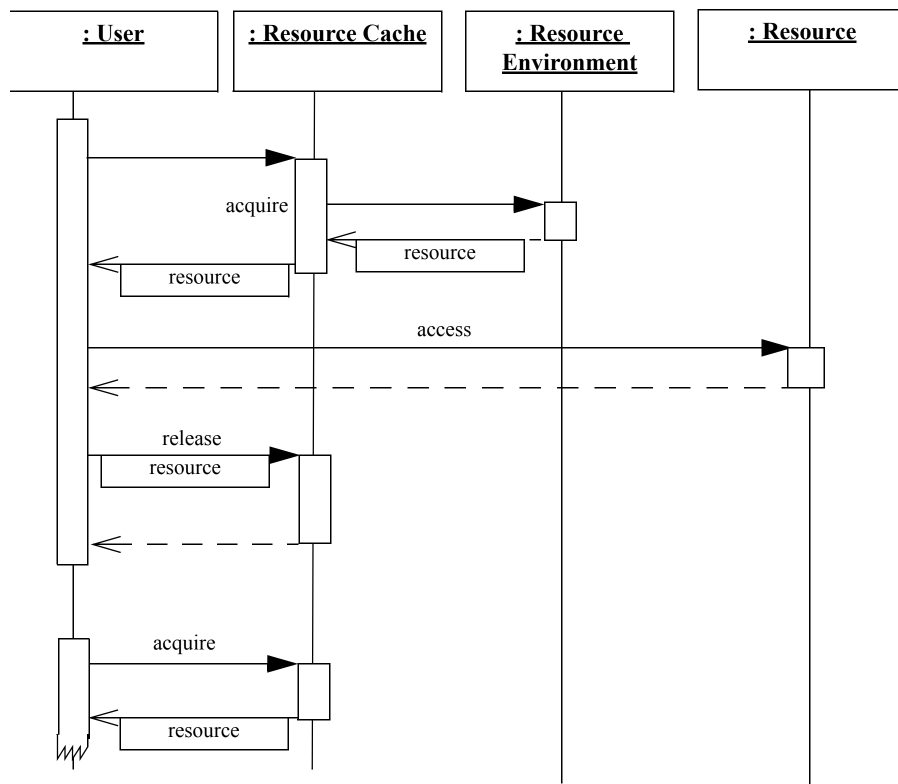
Many resources have states associated with them that must be properly initialized on resource creation. If resource users do not modify resource state, the state of the resources can get lost without losing system information.

In some cases, however, when resources are accessed using write operations, consistency needs to be ensured between the original resource and the resource mirroring it. For details, please refer to the Implementation section.

Dynamics

The following figure shows how the resource user acquires a resource from the resource environment. The resource is then accessed by the user. After the resource has been used it is put into a Cache instead of releasing it to the resource environment.

When the resource user needs to access the same resource again, it uses the Cache to retrieve it. This behavior is depicted in the following sequence diagram.



Implementation

To implement the Caching pattern, the following steps should be followed:

1 *Select resources*: Select resources that benefit from Caching. These are typically resources that are expensive to acquire but used frequently. Caching is very often introduced as an optimization technique after identification of performance bottlenecks.
In distributed systems two forms of Caching can exist: client-side and server-side caching. Client-side caching is useful to save the bandwidth and the time it takes to repeatedly transmit server data to the client. On the other hand, Server-side Caching is useful when many client requests lead to repeated acquisitions of the same resource in the server.

2 *Determine a Caching interface*: When resources are released and re-acquired from the Cache directly by the resouce user, a proper interface must be designed. This interface needs to offer a `release` and an `acquire` method.

```
public interface Cache {
    public void release (Resource resource);
```

```
    public Resource acquire (Identity id);
}
```

The interface above relies on the availability of a separate ID; this might not be the case for every resource. In some cases the ID of a resource might need to be calculated from constituents of the resource.

The `release` method is called by the resouce user when it releases the resource to the Cache instead of releasing it to the resource environment.

3 *Implement the Cache*: The following code snipplet shows the implementation of the `release` method that is called on resource release by the resource user. It adds the resource to the map so that a later call on `acquire` can find it via its ID. For optimization reasons it is advisable to use a hash map since that can perform lookup in almost constant time. The Comparand pattern [CoHa01] gives some ideas on how to perform comparisons between IDs.

```
public class CacheImpl implements Cache {
    public void release (Resource resource) {
        String id = resource.getId ();
        map_.put (id, resource);
    }
    // ...

    java.util.HashMap map_;
}
```

Depending on the kind of resource, the ID of the resource has to be first determined. In the case of our example, the resource can identify itself.

The `acquire` method of the Cache implementation should be responsible for looking up the resource from the map based on the ID. When acquisition from the Cache fails, which means the resource with that identity has not been found, the resource should be acquired from the resource environment. This is especially useful if the Cache has to be integrated transparently. In addition, when resource sharing among multiple users needs to be avoided, the entry with the corresponding ID should be removed from the map.
The following piece of code shows an implementation of the `acquire` method.

```
public class CacheImpl implements Cache {
    public Resource acquire (Identity id) {
        Resource resource = map_.get (id);
        if (resource == null)
            resource = resource_factory_.create (id);
        return resource;
    }
    // ...
    ResourceFactory resource_factory_;
}
```

4 *Determine how to hook in the Cache* (optional): If the Cache will be integrated transparently use an Interceptor or a Virtual Proxy to intercept

release and acquisition requests to the resource environment and delegate them to the Cache.

5 *Decide on eviction strategy*: The resources stored in the Cache require memory. When not used for a long time, it becomes inefficient to hold on to those resources.Therefore, use an Evictor [POSA3] to remove resources that are no longer being used. The integration of the Evictor can be done in multiple ways. For example, an Evictor can get invoked in the context of the `release` method or can be regularly invoked by a timer. Of course, this behavior influences overall predictability. In addition, the Evictor can be configured with different strategies such as LRU, LFU, etc. The Strategy [GHJV95] pattern can be used for this.

6 *Ensure consistency*: In cases where the resource mirrors data that are actually held in some storage, consistency between the mirror and the original has to be ensured. When the original can change, callbacks are used to inform the mirror to update its copy. When the mirror can change, most Caches apply a strategy called “writing through”. Using this strategy changes to the mirror are applied to the original and the mirror directly. This functionality is typically implemented by an entity called a Synchronizer. The Synchronizer is thus an important participant of the pattern. Some Caches further optimize this functionality by introducing more complex logic for keeping the original and mirror in the Cache consistent.
Use a Strategy [GHJV95] to decide on when to synchronize. In some cases only special operations, such as write operations, need to get synchronized immediately, whereas in other cases a periodic update might be advisable. Also, synchronization might be triggered by external events, such as updates of the original by other resource users.

In our motivating example, if the physical data at the NE changes, the memory representation of the NE that is cached must change. Similarly, if the user changes a setting of an NE, the change must be reflected at the physical NE.

Example Resolved

Consider the NMS that needs to monitor the state of many NEs. The middle-tier of the NMS will use the Caching pattern to implement a cache of connections to the NEs. On user request for a specific connection, the connection is acquired. When the connection is no longer needed by the application server, it is stored in the cache. Later, when new requests for the resource arrive, acquisition is done from the cache thus avoiding high acquisition cost.

Subsequent connections to other NEs will then be established when the user first accesses them. When the user context switches to another NE, the

connection is recycled in the connection pool. If a user accesses the same NE, the connection will be reused. No delay will occur on first access to those reused connections.

Variants

Variants describe related patterns, that are derived from this pattern by extending or changing the problem and the solution.

Read-ahead Caches — In a situation where repetitious Partial Acquisition [POSA3] is used to acquire resources, the system can be designed efficiently if a Read-ahead Cache is used. The Read-ahead Cache can acquire resources before they are actually used ensuring that the resources are available when needed.

Synchronized Cache — A synchronized Cache has to ensure consistency between the resources in the cache and the original resources (originals), e.g., the data in a persistent store. Such a Cache would need to ensure that the state of the cached resources is synchronized with the original.

Cached Pools — In situations, where the identity of a resource matters, Cached Pools maintain the association between the resource and the resource user that previously used the resource. This is helpful in situations, when a resource is returned to the pool but acquired soon after. A simple solution would be to return the resource immediately to the pool once it is no longer needed. A more sophisticated solution is to cache the resource (and hence not have it lose its identity). The cache would serve as an intermediary storage for the resource. The cache is configured with a time-out; once the time expires, the resource loses its identity and goes into the pool. The advantage is a small optimization -- if the same resource is required and the resource has not yet been returned to the pool, then you avoid the cost of initialization (virtue of caching).

Known Uses

Paging [Tane02] — Modern operating systems keep pages in memory to avoid expensive read from swap space on disk. The pages that are kept in memory can be regarded as being kept in a cache. Only when a page is not found in cache, does the operating system fetch it from disk.

Offline Files — The Offline Files feature in Windows (since Windows 2000) allows the files and directories of a mounted network drive to be available offline. The files and directories are thus cached and synchronized against the original when connected to the network.

Data Transfer Object [Fowl02]. Middleware technologies such as CORBA and Java RMI allow the remote transfer of objects as opposed to the pure remote method invocation on a remote object. The remote object is actually

transferred "by value" between the client and the server, when methods are invoked on them locally. This is done to minimize the number of expensive remote method invocations. The object is held locally in a cache and represents the actual remote object. Though this approach improves performance, synchronization between the local copy and the remote original of the object must be implemented by the user.

Enterprise JavaBeans (EJB) [Sun02b] — Entity Beans of EJB represent database information in the middle-tier, the application server. This avoids expensive data retrieval (resource acquisition) from the database.

Web Browsers — Most popular web browsers such as Netscape and Internet Explorer cache frequently accessed web pages. If a user accesses the same page, the browsers fetch the contents of the page from cache thus avoiding the expensive retrieval of the contents from the web site. Timestamps are used to determine how long to maintain the pages in cache and when to evict them.

Hardware Cache — Almost every central processing unit (CPU) has an associated hardware memory cache associated with it. The cache avoids slow access to the random access memory (RAM). Cache memory is typically faster than RAM by a factor of two.

Object Cache — An Object Cache applies the pattern to the paradigm of object-orientation. In this case. the resources are objects that have a certain cost associated when created and initialized. The Object Cache allows to avoid those expensive operations when the usage by the resource user allows for Caching.

Data Cache — A Data Cache applies the pattern to data. Data is viewed as resource that is in some cases is hard to acquire. For example, the data could include a complex and expensive calculation or some information that needs to be retrieved from a secondary storage. The pattern allows to reuse fetched data to avoid expensive re-acquisition of data when needed again.

iMerge — The iMerge EMS is an element management system for the iMerge VoIP (voice over internet protocol) hardware system that uses SNMP as the communication interface. It uses Caching to optimize visualization and provisioning of lines between network elements.

Consequences

Caching adds some performance overhead due to an additional level of indirection, but overall there is a performance gain since resources are acquired faster.

In detail, there are several **benefits** of using the Caching pattern:

- *Performance* — Fast access of frequently used resources is an explicit benefit of caching. Unlike Pooling, caching ensures that the resources maintain their identities. Therefore, when the same resource needs to be accessed again, the resource need not be acquired or fetched from somewhere; it is already available.
- *Scalability* — Avoiding resource acquisition and release is an implicit benefit of caching. Caching by its nature is implemented by keeping around frequently used resources. Therefore, just like Pooling, Caching helps avoid the cost of resource acquisition and release.
- Usage Complexity — Caching ensures that the complexity to acquire and release resources from a resource user perspective does not increase.

There are some **liabilities** using the Caching pattern:

- *Synchronization Complexity* — Depending on the kind of resource, complexity increases because consistency between the state of the cached resource and the original data, which the resource is representing, needs to be ensured.
- *Durability* — Changes to the cached resource can be lost when the system crashes. However, if a synchronized cache is used, then this problem can be avoided.
- *Footprint* — The run-time footprint of the system is increased as possibly unused resources are cached. However, if an Evictor is used, then the number of such unused cached resources can be minimized.

As Caching reduces the number of releases and re-acquisitions of resources, it reduces the chance of memory fragmentation. This is similar to Pooling.

Caches are not a good idea if the application requires that data is always available on the expensive media. For example, interrupt-driven I/O intensive applications as well as embedded systems have often no hardware memory caches.

A general note on optimizations: Caching should be applied carefully when other means, such as optimizing the acquisition of the resource itself, cannot be further improved. Caching can introduce some complexity, complicating the maintenance of the overall solution. Therefore consider the trade-off between performance and complexity before applying Caching.

See Also

Pooling [POSA3]—The main idea behind Pooling is reuse of resources. It helps avoid the cost of (re-)acquisition and release of resources. Resources are typically anonymous.

Evictor [POSA3]—An Evictor an be used for eviction of cached data.

Virtual Proxy [GHJV95]—A Virtual Proxy can be used to hide caching effects. Smart Proxies in CORBA (TAO) intercepting remote invocations are especially designed for this.

Cache Management [Gran98]—The Cache Management pattern focuses on how to combine a cache with the Manager pattern [MRB98], where the Manager pattern centralizes access, creation and destruction of objects. The description is more specific to objects and database connections, both in the context of Java.

Acknowledgements

Thanks to Ralph Cabrera for sharing his experience on Caching with us and for providing the iMerge known use. Special thanks to Pascal Costanza, our EuroPLoP 2003 shepherd, for his excellent comments and patience.

References

[CoHa01] P. Costanza and A. Haase, *The Comparand Pattern*, European Conference on Pattern Languages of Programs, Kloster Irsee, Germany, July 2001

[Fowl02] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002

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

[Gran98] M. Grand, *Patterns in Java - Volume 1*, John Wiley and Sons, 1998

[MRB98] R. C. Martin, D. Riehle, and F. Buschmann, eds., *Pattern Language of Program Design 3*, Addison-Wesley, 1998

[OMG02] OMG, *Common Object Request Broker Architecture*, http://www.omg.org, 2003

[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

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

[Sun02a] Sun, *Java Remote Method Invocations (RMI)*, http://java.sun.com/products/jdk/rmi/, 2003

[Sun02b] Sun, *Java 2 Enterprise Edition (J2EE)*, http://java.sun.com/j2ee/, 2003

[Tane02] A. S. Tanenbaum, Computer Networks, Fourth Edition, Prentice Hall, 2002

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