Patterns for Resource Management

Iuliana Bocicor
iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2018

Overview

Resources in the real world I

- A **resource** is an asset that is available in a limited supply and which produces a certain benefit.

Figure source: http://mailemedicinals.com/content/lake-mountains-view-wallpapers.html

Resources in the real world II

- Examples of resources in the real world:
  - Natural resources: water, soil, minerals (gold, iron, coil, petroleum), forests.
  - Biological resources: nutrients, sunlight.
  - Economic resources: capital, human resources (skills, knowledge, abilities).

Resources in software systems I

- Computer resources are physical or virtual components of limited availability within a computer system.
- Examples: memory, CPU, file handles, network connections, database sessions.
- Usually, a *resource user* (*requestor*) needs the resource to perform a specific task.
- The *resource provider* is the mechanism that provides the resource on request.

Resources in software systems II

- Resources can be:
  - *Reusable* - acquired, used, released and then used again. E.g.: memory, file handles.
  - *Non-reusable* - are consumed and cannot be reused. E.g.: processing time in a computer grid.
  - *Concurrent* - can be used concurrently by multiple users, thus need synchronization. E.g.: database.
  - *Exclusive* - can only be used by a single user. E.g.: processing time of a service.

Resource management I

- Resource management refers to the process by which the availability of resources is controlled and how these are provided to resource users.
- Efficient resource management is quite difficult: resources must be made available when needed, they must have a deterministic lifecycle and must be released correctly.
- Resource management has a strong impact on non-functional system requirements such as performance, scalability, flexibility or stability.

Resource management II

- Correct resource management should be addressed early in the development lifecycle of a software system.
- Resource management is important all types of applications: embedded, large enterprise systems, grid computing.
- It can be implemented in all abstract layers of a software system (middleware, service layer, UI layer).

Resource Acquisition I

- Resource acquisition can be achieved in various ways and at different times.
- Before a resource is acquired, it has to be found - the *Lookup* pattern.
- The moment of acquisition can influence the system's performance and scalability:
  - *Lazy acquisition* refers to delaying the moment until the resource is actually needed, to avoid unnecessary exhaust.

Resource Acquisition II

- *Eager acquisition* means acquiring the resource as early as possible, in case of real-time constraints within the system.
- *Partial acquisition* refers to acquiring only a part of a resource, in case the resource is large and not needed in its entirety at all times.

Lookup I

Figure source: http://acomputerscientistlearnsaws.blogspot.ro/2017/05/chapter-11-aws-directory-service.html

Lookup II

- Provides a way of finding and accessing local or distributed resources in a system.

Problem

- How can resource providers and resource users interact efficiently to ensure that users can find and access resources published by providers, considering that some resources may be added or removed over time?

Lookup - Solution

- **Lookup service** - a "central point of communication" between providers and users.
- Resource providers advertise resources together with their properties and resource users can search resources and further uses them.
- Resource providers and resource users do not need to know about each other's location.

Lookup - Structure

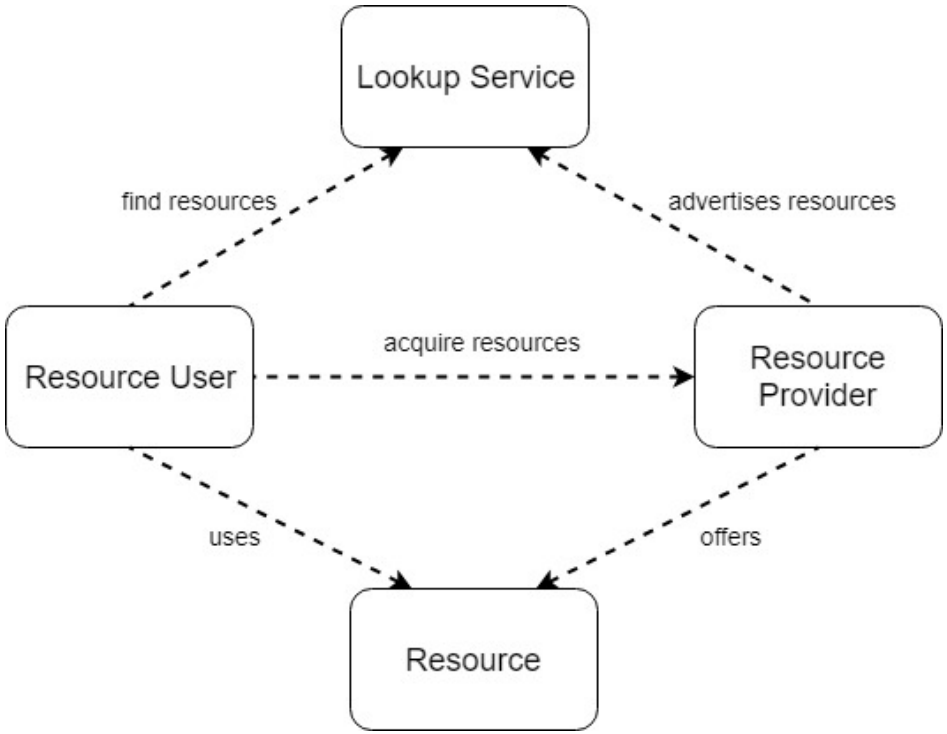


Figure adapted from: Michael Kircher, Prashant Jain, Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource Management, Wiley, 2004.

Lookup - Implementation I

- *Create an interface for the lookup service:*
  - Allow resource providers to register and unregister resource references, together with resource associated properties.
  - Facilitate advertisement and lookup of resources.
- *Implement the lookup service:* registered references and their information may be kept in a hash map or a tree structure or a persistent repository.

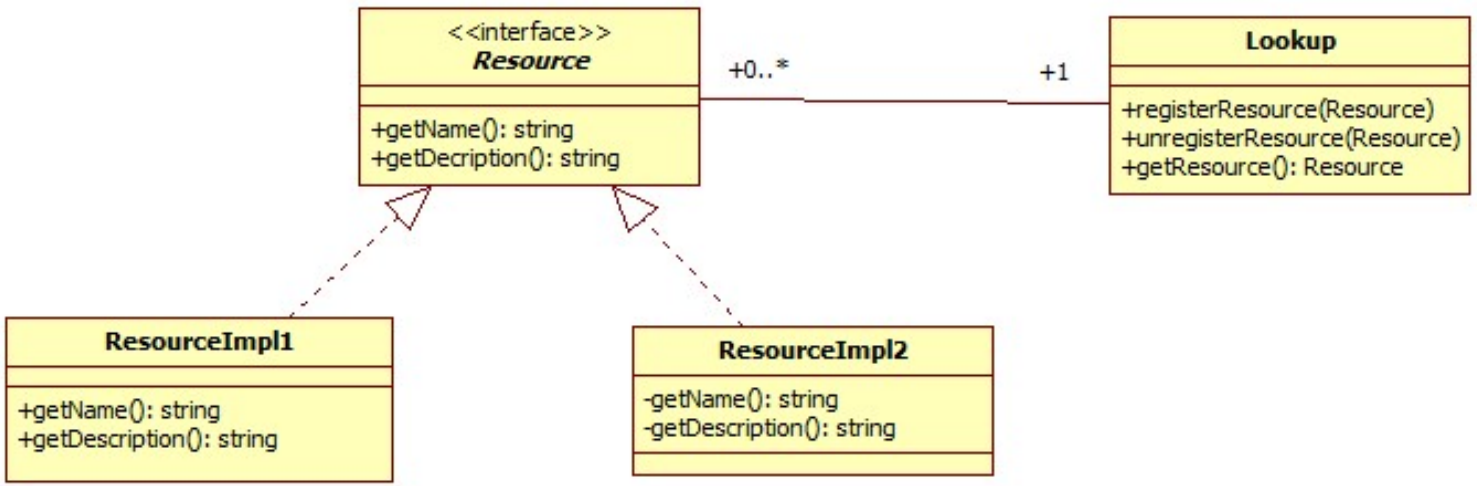
Lookup - Implementation II

- *Lookup service access point:* host name and port number where the lookup service is running must be provided via properties/configuration files or environment variables.
- *Query language (optionally):* used to allow resource users to search resources via complex queries.

DEMO

Lookup example (*Lookup.h*, *Lookup.cpp*).

Lookup - Implementation III



Lookup - Consequences and Uses I

Advantages

- Resource users can find available resources, according to resource properties.
- In distributed environments a bootstrapping protocol can be used by a resource user to find other distributed services via the lookup service.
- Resource providers and resource users are independent, they do not need to know about each other's location.
- Little or no manual configuration is needed in distributed systems.

Lookup - Consequences and Uses II

Disadvantages

- If the lookup service crashes, all resources will need to be re-registered on restart (unless persistence is involved).
- Danger of dangling references - if a resource is no longer available, but its reference has not been removed from the lookup service.

Practical Uses

- JNDI (Java Naming and Directory Interface);
- COM+ (COM registry keys);
- DNS (Domain Naming Service);
- Grid Computing.

Lazy Acquisition I

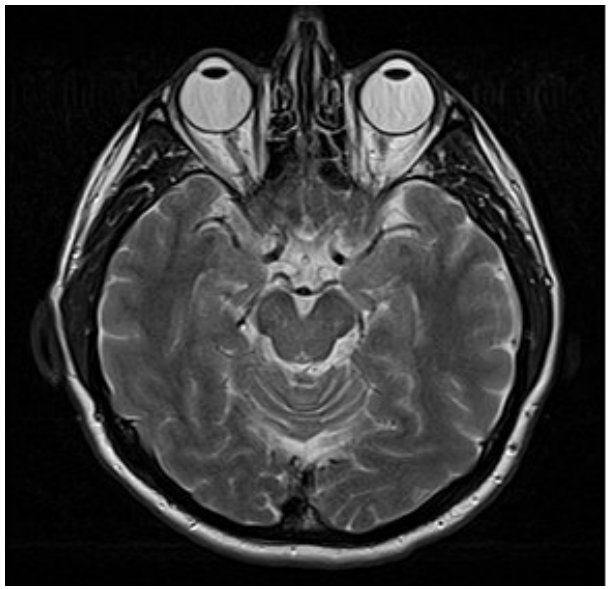


Figure sources: https://en.wikipedia.org/wiki/Magnetic_resonance_imaging_of_the_brain,https://www.imedicalapps.com/specialty/radiology/

Lazy Acquisition II

- To optimize resource use, this pattern delays resource acquirement to the latest possible time.

Problems

- Limited resource availability in a system may cause bottlenecks and impact system performance.
- Acquiring resources when the system starts is not always a good idea (may cause acquisition overhead), especially if the resources are not needed immediately.

Lazy acquisition - Solution

- Resources are acquired at the latest possible time, when it is no longer possible to postpone its acquisition.
- A resource proxy is created at initial resource request.
- When resource access is necessary, the resource proxy acquires the actual resource and provides it to the resource user.
- The resource proxy's interface is identical to the resource interface, thus the user is oblivious to the indirection.

Lazy acquisition - Structure

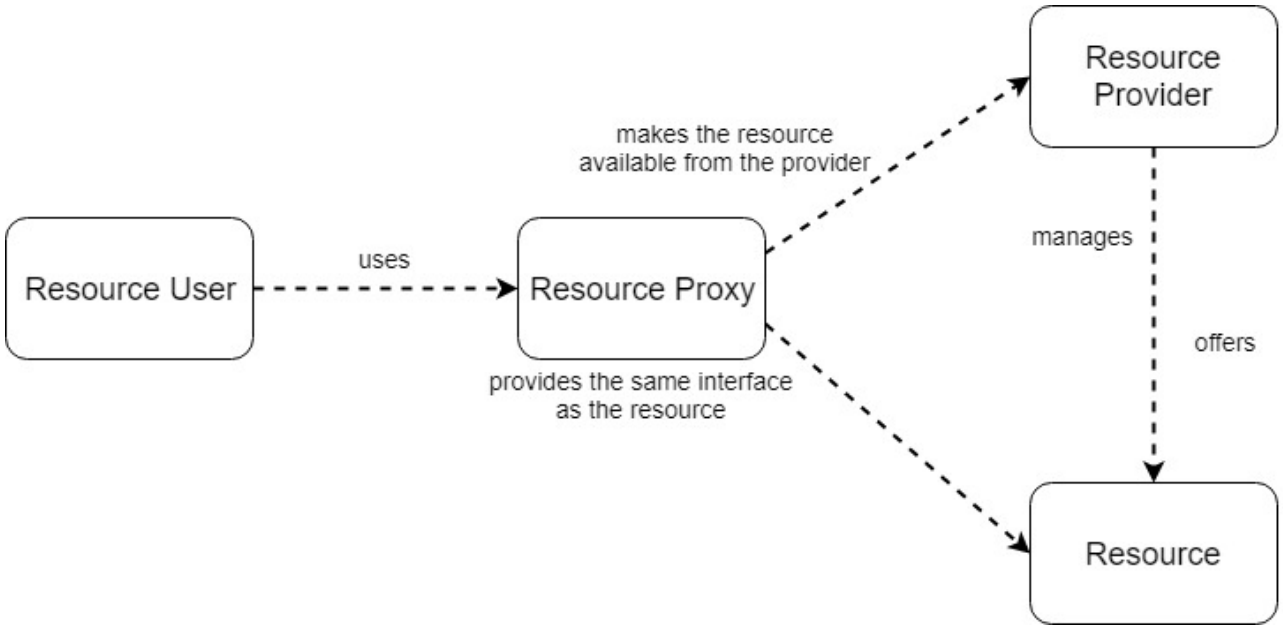


Figure adapted from: Michael Kircher, Prashant Jain, Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource Management, Wiley, 2004.

Lazy acquisition - Implementation I

- *Identification of resources that are suitable for lazy acquisition:*
  - are expensive to acquire.
  - are not needed immediately.
- *Definition of the resource proxy interface:* should be identical to the resource interface.
- *Implementation of the resource proxy:* it must hide the lazy acquisition of the resource such that whether the resource proxy or the resource is accessed is transparent to the user.

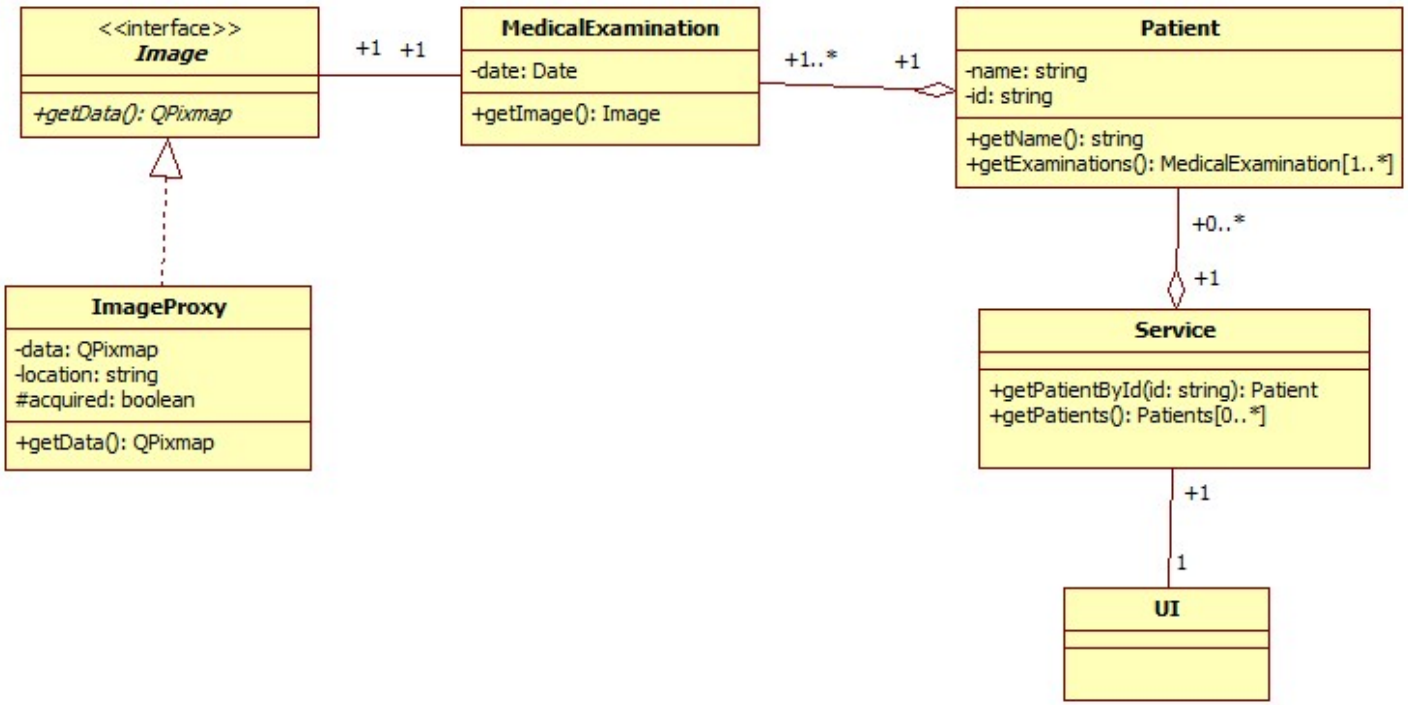
Lazy acquisition - Implementation II

- *Definition of the acquisition strategy*, e.g.:
  - the resource is acquired only when accessed;
  - the resource is acquired when some an associated resource is being used and its usage will trigger the need for the current resource.

DEMO

Lazy acquisition example (*project LazyAcquisition*).

Lazy acquisition - Implementation III



Lazy acquisition - Consequences and Uses I

Advantages

- If not all resources are acquired at start-up:
  - the system start-up time is optimized.
  - the system will not face the danger of exhaustion and will be more stable.
- The entire process is transparent to the user. The user is not aware that acquisition is made via the proxy.

Lazy acquisition - Consequences and Uses II

Disadvantages

- Slight memory overhead (memory for proxies).
- The additional level of indirection may induce delays and overhead during regular progra execution.

Practical Uses

- Singleton objects (lazy instantiation).
- Operating systems (library loading).
- Haskell (lazy evaluation of expressios).
- Hibernate ORM (lazy loading).

Eager Acquisition I

Figure source: https://www.elprocus.com/real-time-applications-of-embedded-systems/

Eager Acquisition II

- This pattern describes how run-time acquisition of resources can be optimized, by acquiring them eagerly, before their use.

Problems

- Expensive resource acquisition at run-time may cause unpredictable time overheads.
- Resource acquisition should be both fast and predictable (the amount of time for each resource acquisition should be the same).

Eager acquisition - Solution

- Resources are acquired from the resource provider before their actual use by a provider proxy.
- The provider proxy stores the resources in a container.
- When resource access is needed by the resource user, the provider proxy returns the resource from the container.
- The time of acquisition by the proxy can be at system start-up or at a specific calculated time after start-up, according to the system's needs.

Eager acquisition - Structure

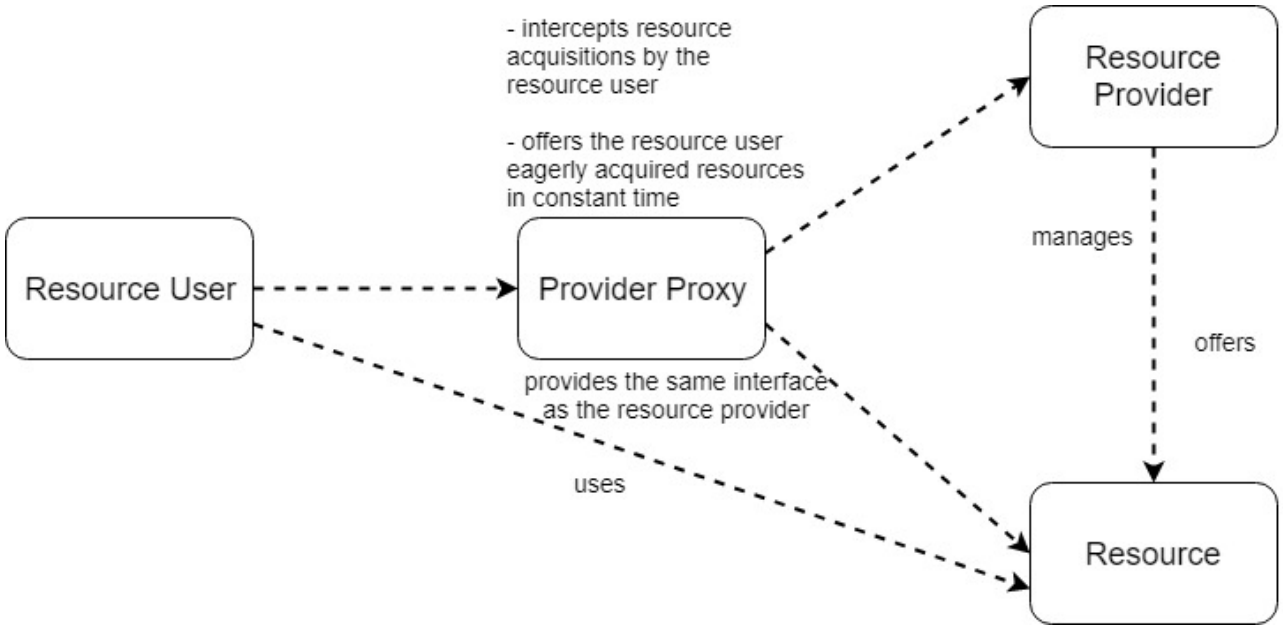


Figure adapted from: Michael Kircher, Prashant Jain, Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource Management, Wiley, 2004.

Eager acquisition - Implementation I

- *Identification of resources that are suitable for eager acquisition:*
  - are expensive to acquire (connections, memory).
  - must be made available fast.
- *Implementation of the provider proxy:* it will have the same interface as the resource provider. This step can also be skipped when transparency to the resource user is not required.

Eager acquisition - Implementation II

- *Implementation or use of a container* to hold the eagerly acquired resources.
- *Identification of the timing strategy*, e.g.:
  - at system start-up;
  - at specific (calculated) times during run-time.

DEMO

Eager acquisition example (*project EagerAcquisition*).

Eager acquisition - Consequences and Uses I

Advantages

- Resource acquisition is predictable and very fast (short, constant time).
- Acquisition of resources by the provider proxy can follow a unified, configurable strategy, thus avoiding side effects (e.g. memory fragmentation).
- The process is transparent to the user.

Eager acquisition - Consequences and Uses II

Disadvantages

- Eagerly acquired resources must be properly managed.
- The number of need resources must be estimated beforehand.
- If too many resources are acquired eagerly, this can lead to system exhaustion.
- System start-up is slower.

Eager acquisition - Consequences and Uses III

Practical Uses

- Ahead of time compilation (C++, Java).
- Hibernate ORM (eager loading).
- Eclipse plug-ins (declarations that determine visualization).
- Hamster (the animal :)).

Resource Lifecycle I

- After acquisition, resource lifecycle must be managed efficiently.
- Managing a resource's lifecycle refers to making it available to resource users, managing other dependent resources and releasing it, when it is no longer necessary.
- *Caching* and *Pooling* patterns are used to optimize the acquisition and release of reusable resources, that are used serially. Caching maintains the identity of resources, while pooling does not.

Resource Lifecycle II

- When two or more resources / resource users / providers interact to change the system, keeping the system in a consistent state is achieved via the *Coordinator* pattern.
- The *Resource Lifecycle Manager* pattern takes the responsibility of resource management, thus freeing the resource users or resources of this burden.

Caching I

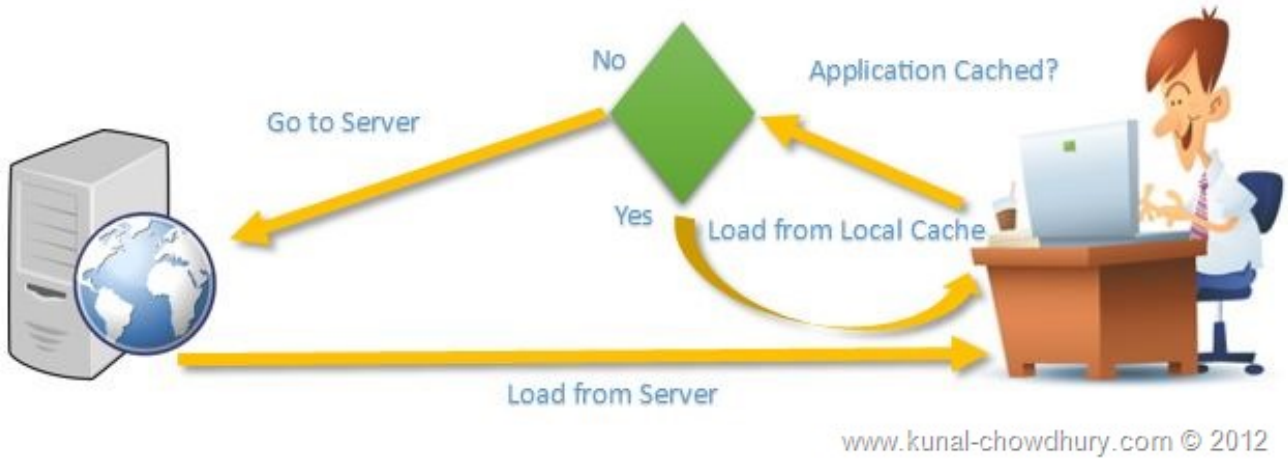


Figure source: https://www.cbronline.com/what-is/what-is-cache-memory-4930080/

Caching II

- This pattern describes how to store resources and their identities in a fast-access storage, to be able to use them at a later time, without the need of expensive re-acquisition.

Problems

- Repeatedly acquiring, initializing and releasing the same resource is expensive (time, overall performance) causes overhead.
- If resource providers are temporarily unavailable, resources are not accessible.

Caching - Solution

- Resources are temporarily stored in a cache, from where they can be accessed quickly.
- In this manner, resources no longer have to be re-acquired from the resource provider each time they are needed.
- Resources in the cache are identified by a unique identifier.
- When resources are no longer needed, they are evicted.

Caching - Structure

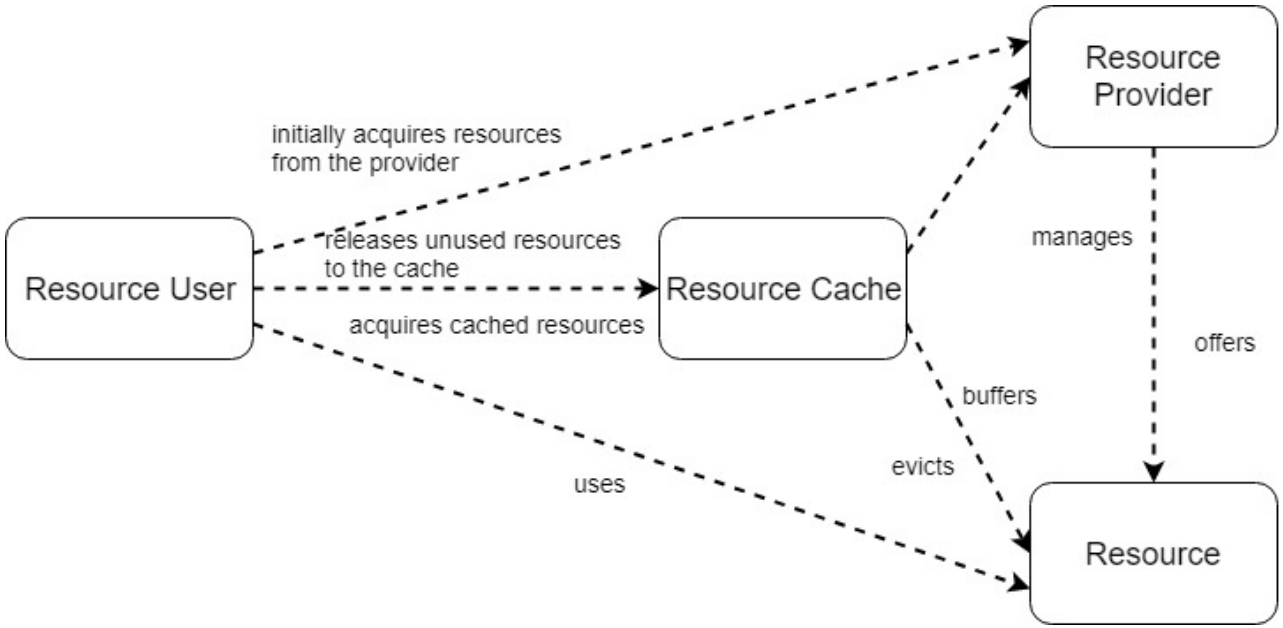
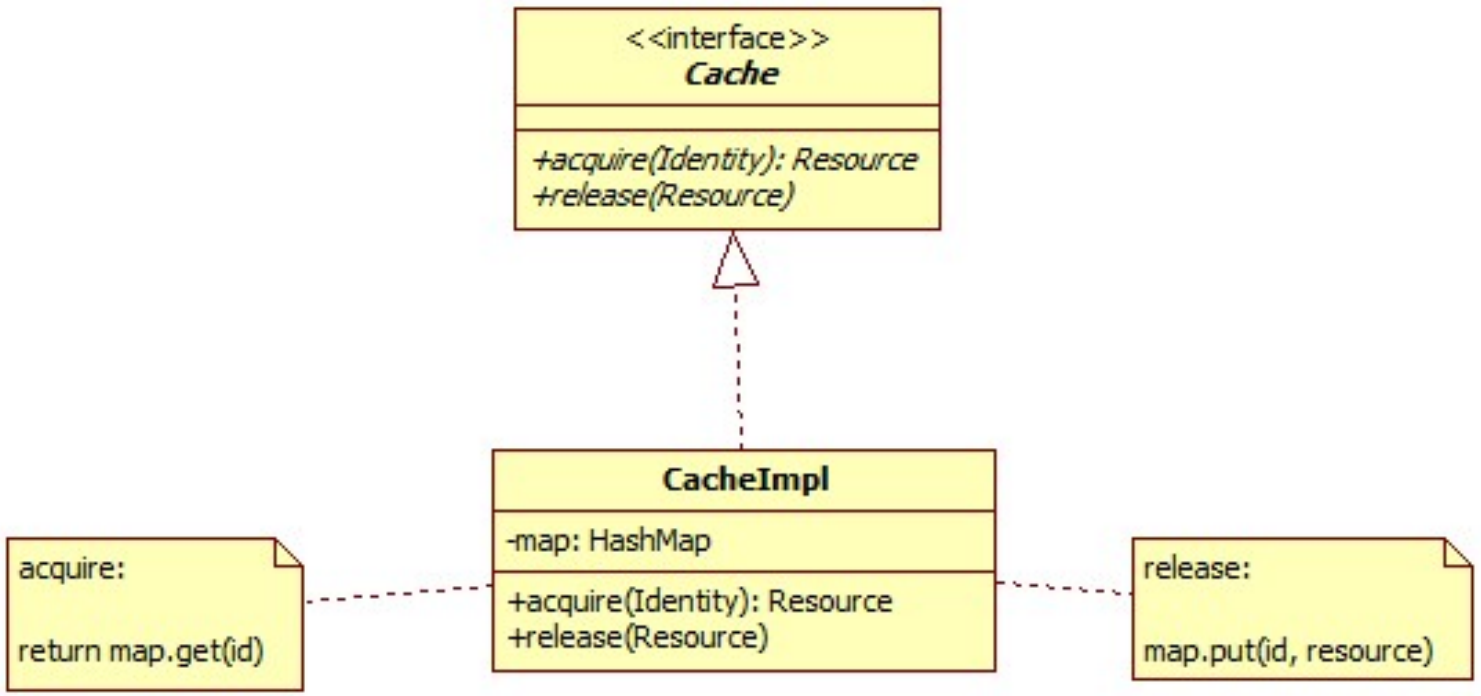


Figure adapted from: Michael Kircher, Prashant Jain, Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource Management, Wiley, 2004.

Caching - Implementation I

- *Identification of resources that should be cached*:
  - expensive to acquire;
  - used frequently.
- *Determine a caching interface*: at least operations for acquiring and releasing resources.
- **Implement the cache** (operations implementation).
- *Decide on an eviction strategy* to remove resources that are no longer being used.
- Ensure synchronization between original resources and cached resources (when resources are being modified).

Caching - Implementation II



Caching - Consequences and Uses I

Advantages

- Resources are available when needed (fast access to frequently used resources).
- Resources are available even when resource providers are temporarily unavailable.
- Greater system stability, due to a reduced number of releases and re-acquisition of resources.

Caching - Consequences and Uses II

Disadvantages

- Ensuring synchronization between original and cached resources might become very complex (e.g. clustered environments).
- At a system crash, changes to the cached resources could be lost.
- If unused resources are cached, system run-time could be increased.

Caching - Consequences and Uses III

Practical Uses

- Hardware cache (CPU).
- Web browsers.
- Web proxy.
- Paging (operating systems).

Pooling I

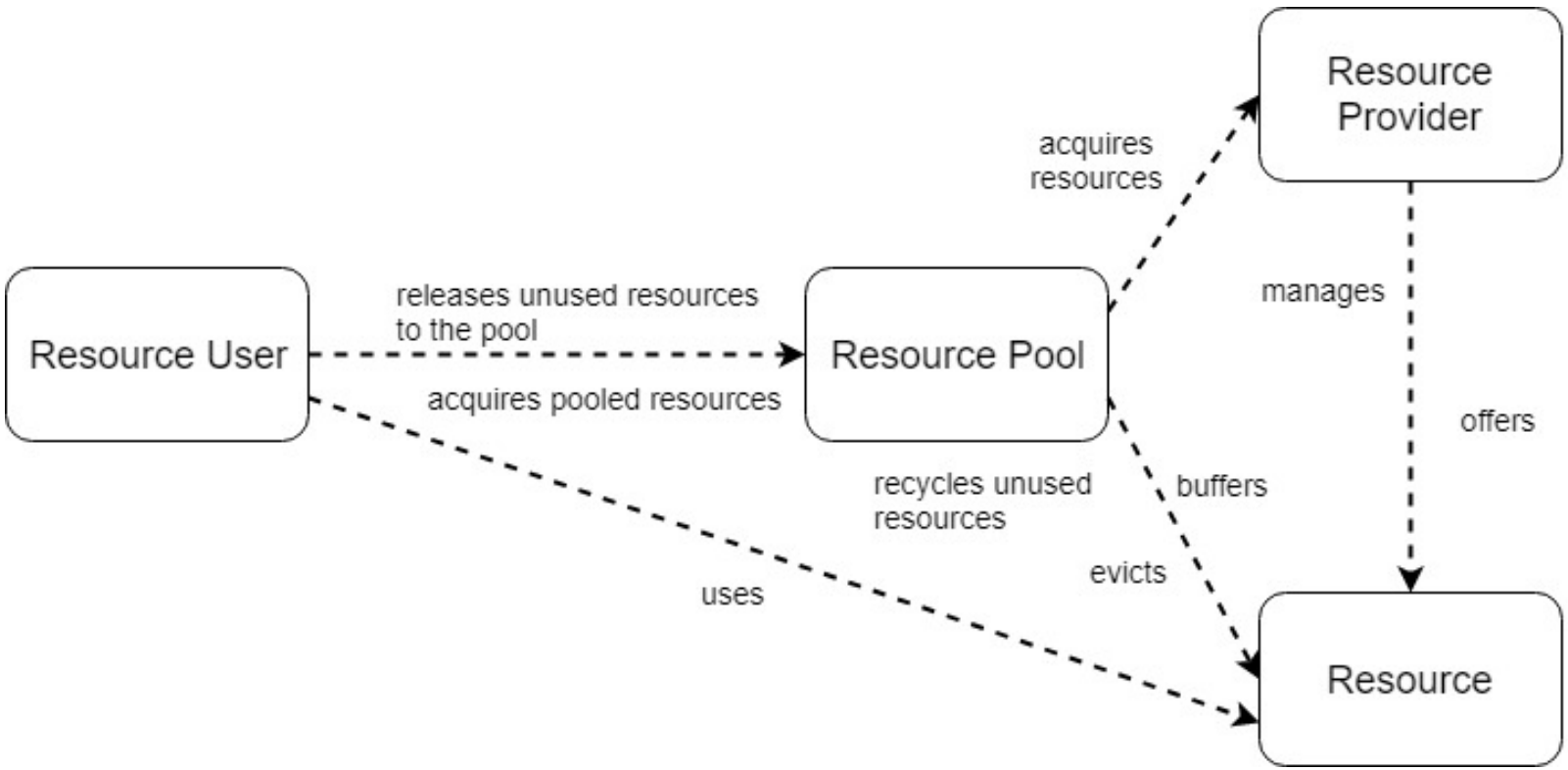


Figure adapted from: http://slideplayer.com/slide/6960288/

Pooling II

- This pattern describes how to recycle and reuse expensive resources. When a resource is recycled and pooled, it looses its identity and state.

Problems

- Re-acquisition and release of resources, in a repeated manner causes:
  - time wastage;
  - overhead;
  - application complexity;
  - system instability (e.g. memory fragmentation on operating systems).

Pooling - Solution

- A pool is used to manage multiple instances of one type of resource.
- Released resources (when they are no longer needed by resource users) are put back into the pool.
- Before reuse, the resource will have to be initialized.
- Resource identity is not used for resource identification in the pool.

Pooling - Structure

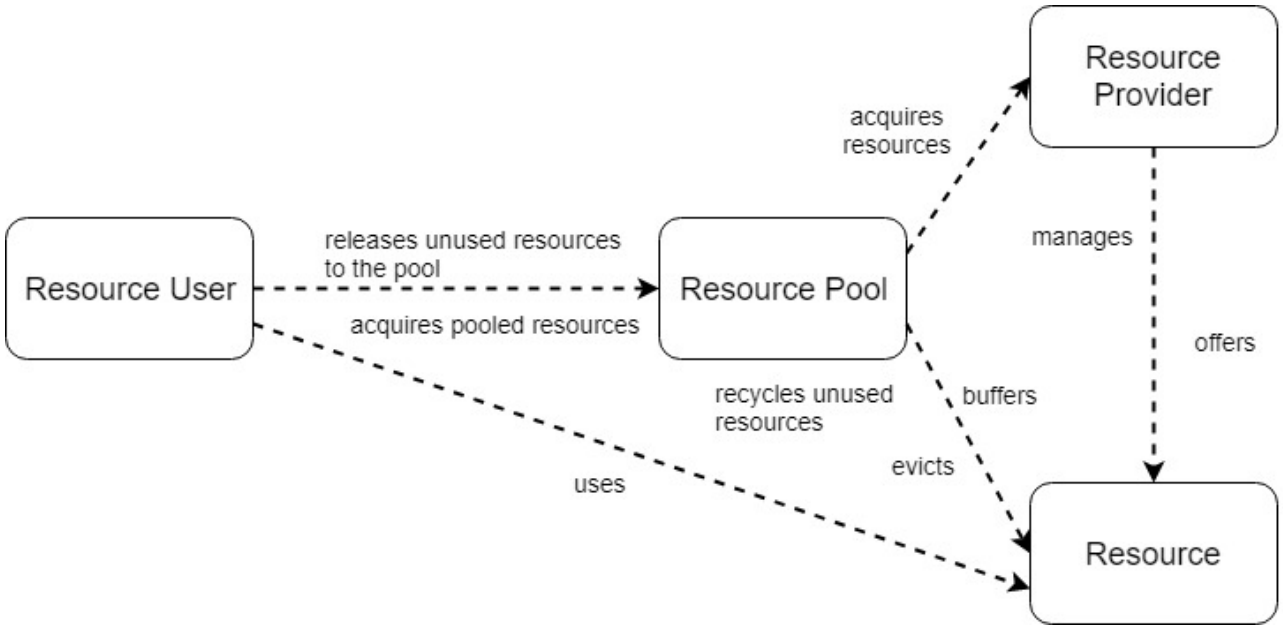


Figure adapted from: Michael Kircher, Prashant Jain, Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource Management, Wiley, 2004.

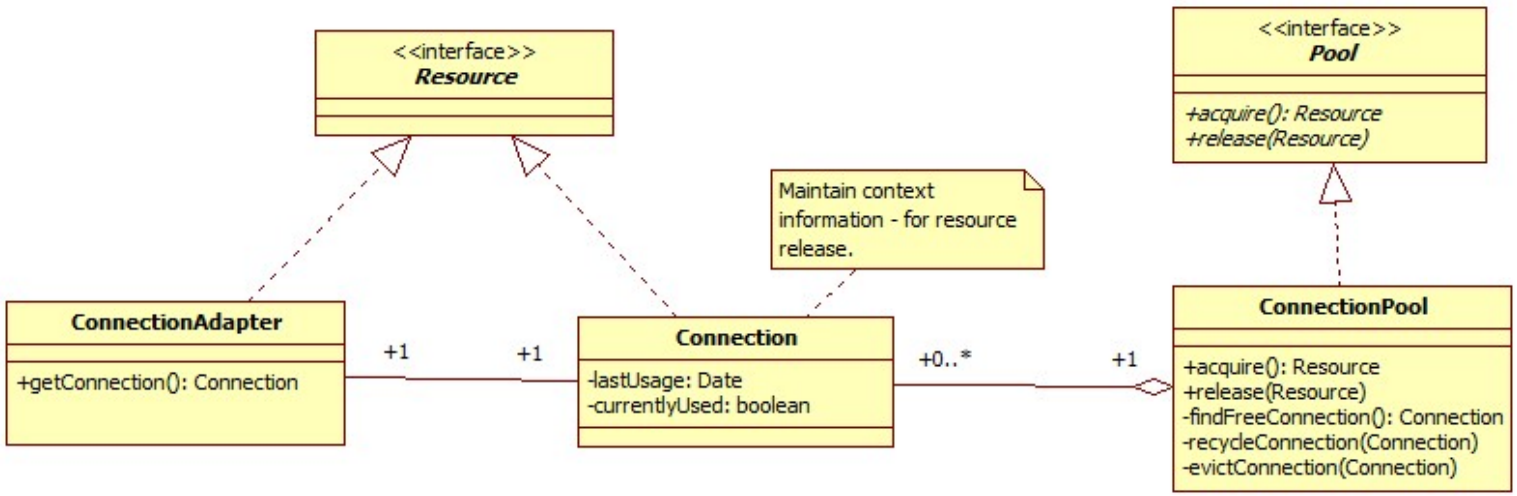
Pooling - Implementation I

- *Identification of resources that should be pooled.* Resources should be grouped in different pools, according to their types.
- *Set a maximum size for the pool:* to avoid resource exhaustion.
- A certain number of resources that are more frequently used should be acquired eagerly, to minimize resource acquisition time at run-time.
- *Define a resource interface*, which must be implemented by all pooled resources.

Pooling - Implementation II

- Any implementation of the interface will retain context information needed to know when and how the resource must be evicted.
- *Define a pool interface:* operations for the acquisition and release of resources.
- An implementation of this interface will have to store a container of resources.
- Some of the unused resources from the pool should be released from time to time to avoid wasting space and degrading performance.

Pooling - Implementation III



Pooling - Consequences and Uses I

Advantages

- Time of acquiring expensive resources is reduced, therefore performance is imprived.
- Predictability of resource acquisition.
- Increase of system stability, due to recycling and avoidance of costly release and re-acquisition of resources.
- Unused resources from the pool can be shared between resource users.

Pooling - Consequences and Uses II

Disadvantages

- Resource users have to release resources back to the pool.
- Synchronization in concurrent environments.

Practical Uses

- Web servers - thread pooling.
- DB connection pooling.

Differences between caching and pooling

	Cache	**Pool**
Request by the user	Specific object in the cache	Any object in the pool
All objects are used	A request must wait for the specific object to be released, if it is used.	When any object is returned to the pool, the request can be satisfied.
Size	Fixed (an object has to be removed to allow a new object into the cache).	Fixed or can grow. New objects can be added.

Resource Release I

- Unused resources of a system should be released to avoid resource starvation, to maintain system stability and to increase scalability.
- The *Leasing* pattern allows defining a lease for a resource, where the grantor is the resource provider and the holder is the resource user. The lease also specifies a time duration. When the lease expires and is not renewed, the resource is released.
- The *Evictor* pattern refers to determining which resources need to be released and when.

Evictor I

Figure source: https://www.irisns.com/how-to-learn-for-the-network-management-system-of-the-future/

Evictor II

- This pattern describes how, when and which resources should be released.

Problems

- Some resources are only seldom used (some even just one time).
- If resources are not released, the entire performance of the system could be affected.
- If resources are released after usage, those that are frequently used will have to be reacquired. This might be expensive, for frequently used resources.

Evictor - Solution

- Resources' lifetimes are monitored and controlled.
- When a resource is being used by the system, it is marked.
- Resources that are not frequently or recently used are not marked.
- From time to time, non-marked resources are evicted (released).

Evictor - Structure

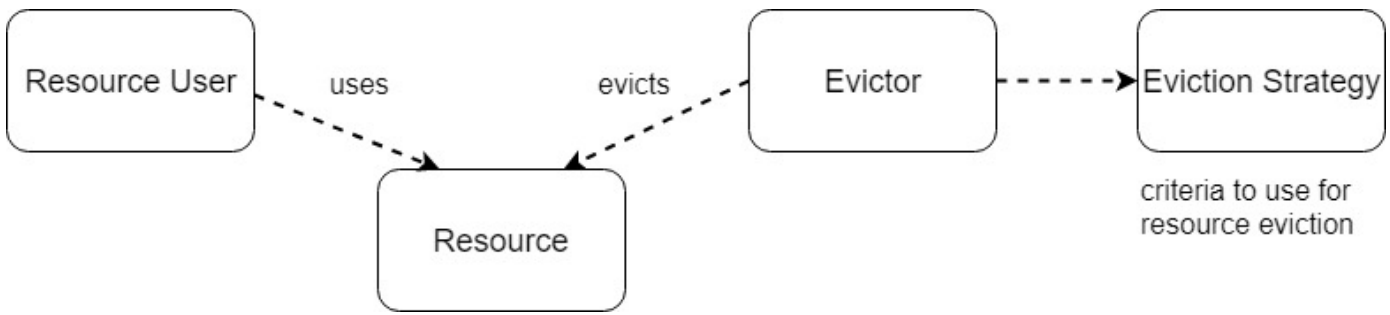


Figure adapted from: Michael Kircher, Prashant Jain, Pattern-Oriented Software Architecture, Volume 3, Patterns for Resource Management, Wiley, 2004.

Evictor - Implementation I

- *Definition of an Eviction interface:*
  - Function to determine if the object is evictable.
  - Function to determine object specific information to verify the evicting criteria.
  - Function that defines the clean-up that needs to be done before evicting (e.g. if the object has acquired other resources or if it has to persist its state).
- *Identification of evictable resources:* resources that cannot be re-acquired or those that are very frequently used should **not** be evicted.

Evictor - Implementation II

- Definition of an eviction strategy:
  - Least Recently Used (LRU)
  - Least Frequently Used (LFU)
- Add business logic for evicting.

DEMO

Evictor example (*Evictor.h*, *Evictor.cpp*).

Evictor - Consequences and Uses I

Advantages

- If the system keeps an upper limit on the number of resources that are being used at any time, the application is easily scalable, with no impact on memory.
- Only essential resources are kept in memory $\Rightarrow$ efficient application.
- Resource release is transparent to the user. The user does not have to handle this task.
- If any problems or errors might arise, the proper disposal of resources is guaranteed.

Evictor - Consequences and Uses II

Disadvantages

- Additional business logic is needed to determine which resources should be evicted and when.
- Resource eviction leads to execution overhead.
- Resource re-acquisition is expensive (this might be needed if the eviction strategy is not suitable and resources which were evicted are needed again).

Practical Uses

- Enterprise JavaBeans (EJB).
- .NET and Java (garbage collector).
- Paging in operating systems.

Summary I

- Resource management refers to the process by which the availability of resources is controlled and how these are provided to resource users.
- Correct resource management should be addressed early inthe development lifecycle of a software system.
- Resource acquisition, resource lifecycle, resource release.

Summary II

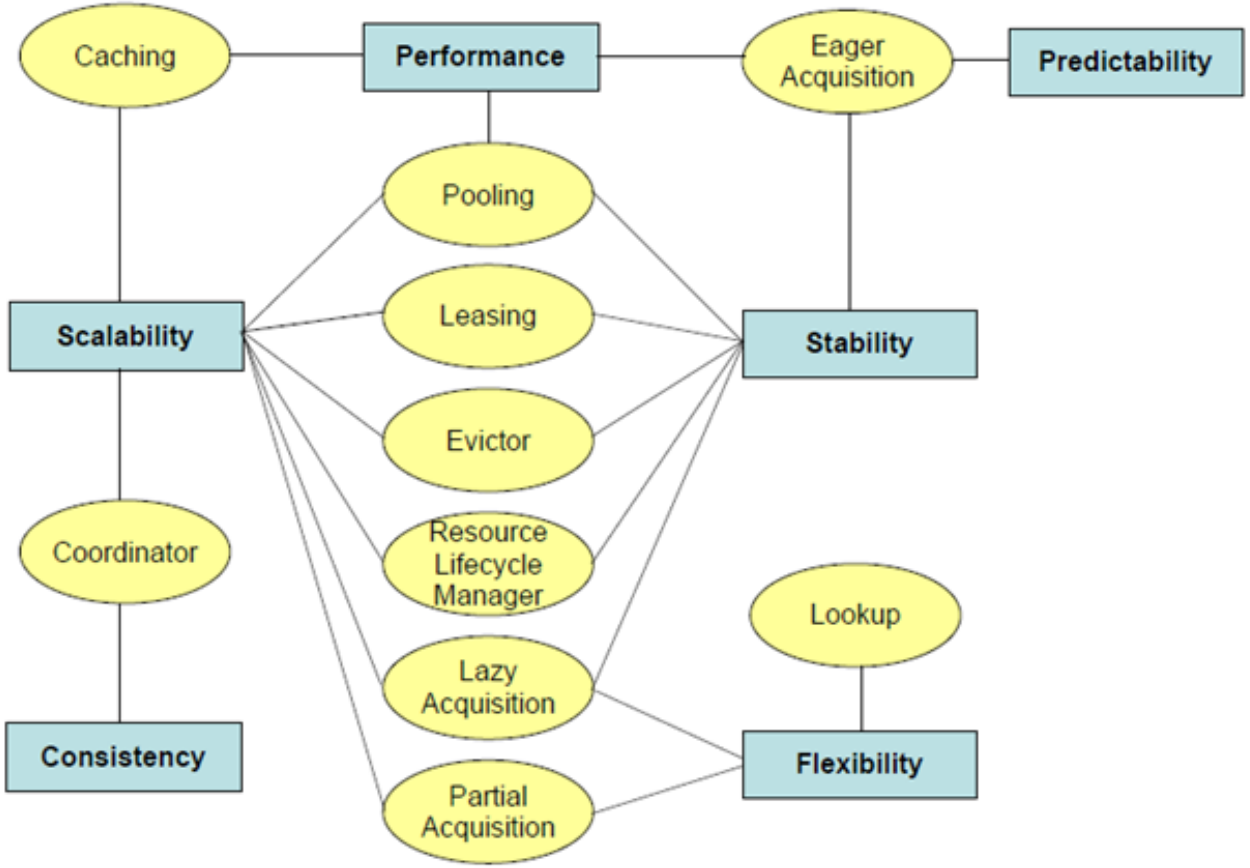


Figure source: https://intelrendz.wordpress.com/tag/resource-management-design-patterns/

Reeferences I

- Michael Kircher, Prashant Jain. *Pattern-Oriented Software Architecture*, Volume 3, Patterns for Resource Management, Wiley, 2004.
- Michael Kircher, *Lazy Acquisition*, Siemens AG, 2002.
- Michael Kircher, *Eager Acquisition*, Proceedings of the 7th European Conference on Pattern Languages of Programms (EuroPLoP '2002), Germany, 2002.
- M. Kircher and P. Jain, *Pooling Pattern*, Proceedings of the 7th European Conference on Pattern Languages of Programms (EuroPLoP '2002), Germany, 2002.

Reeferences II

- D. Bellebia, J-M. Douin. *Applying Patterns To Build A Lightweight Middleware For Embedded Systems*, Proceedings of the 2006 conference on Pattern languages of programs, 2006.
- Markus Aleksy, Ralf Gitzel, Gerhard Vollmar, Nicolaie Fantana, Christian Stich. *Techniques For The Efficient Resource Management Of Contextsenstivie Mobile Applications And Their Utilization In Industrial Field Service*, Journal of Mobile Multimedia, Vol. 4, No.3 and 4 (2008) 200-209.
- Prashant Jain, Evictor, Siemens AG, Corporate Technology, 2001.