2 Resource Acquisition

"I find that a great part of the information I have was acquired by looking up something and finding something else on the way."

Franklin P. Adams

The lifecycle of a resource begins with its acquisition. How and when resources are acquired can play a critical role in the functioning of a software system. By optimizing the time it takes to acquire resources, system performance can be significantly improved.

Before a resource can be acquired, the most fundamental problem to be solved is how to find a resource. The Lookup (21) pattern addresses this problem and describes how resources can be made available by resource providers, and how these resources can be found by resource users.

Once a resource has been found, it can be acquired. The timing of resource acquisition is important, and is mainly addressed by Lazy Acquisition (38) and Eager Acquisition (53). While Lazy Acquisition

defers acquisition of resources to the latest possible point in time, Eager Acquisition instead strives to acquire resources as early as possible. Both extremes are important and are heavily dependent on the use of the resources.

If resources that are acquired are not used immediately it can lead to their wastage. Lazy Acquisition addresses this problem and consequently leads to better system scalability. On the other hand, some systems have real-time constraints and stringent requirements for the timing of resource acquisition. Eager Acquisition addresses this problem, and consequently leads to better system performance and predictability.

A resource of large or unknown size may not be needed completely, and therefore it might be sufficient to acquire only part of the resource. The Partial Acquisition (81) pattern describes how the acquisition of a resource can be partitioned into steps to allow only part of the resource to be acquired. Partial Acquisition can serve as a bridge between Lazy Acquisition and Eager Acquisition. The first step of Partial Acquisition typically acquires part of the resource eagerly, while subsequent steps defer further resource acquisition to a later stage.

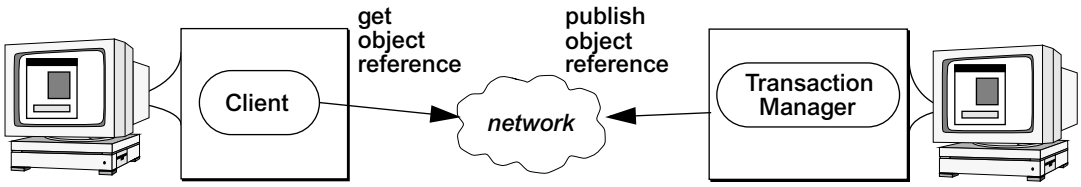
The Lookup pattern can be used with both reusable and non-reusable resources. Furthermore, it can support both concurrently accessible resources and exclusive resources. This is because Lookup only focuses on providing access to resources, and does not deal with how the resources are actually used.

Lazy Acquisition, Eager Acquisition and Partial Acquisition can also be used with both reusable and non-reusable resources. However, if a reusable resource is concurrently accessible, the three patterns can provide certain optimizations such that the resource is only acquired once and can then be shared by the concurrent users. Such an optimization requires special handling in the case of Partial Acquisition. This is because the amount of a resource that is partially acquired can vary among the concurrent users.

Lookup

The *Lookup* pattern describes how to find and access resources, whether local or distributed, by using a lookup service as a mediating instance.

Example Consider a system that consists of several services implemented as remote objects using CORBA [OMG04a]. To access one of the distributed services, a client typically needs to obtain a reference to the object that provides the service. An object reference identifies the remote object that will receive the request. Object references can be passed around in the system as parameters to operations, as well as the results of requests. A client can therefore obtain a reference to a remote object in the system from another object. However, how can a client acquire an initial reference to an object that the client wants to access?



For example, in a system that provides a distributed transaction service, a client may want to obtain a reference to the transaction manager so that it can participate in distributed transactions. How can a server make the transaction manager object reference that it created widely available? How can a client obtain the transaction manager object reference without having a reference to any other object?

Context Systems where resource users need to find and access local and distributed resources.

Problem Resource providers may offer one or more resources to resource users. Over time additional resources may be added or existing resources may be removed by the resource provider. One way the

resource provider can publish the availability of existing resources to interested resource users is by periodically sending a broadcast message. Such messages need to be sent on a periodic basis to ensure that new resource users joining the system become aware of available resources. Conversely, a resource user could send a broadcast message requesting all available resource providers to respond. Once the resource user receives replies from all available resource providers, it can then choose the resource(s) it needs. However, both of these approaches can be quite costly and inefficient, since they generate lots of messages, which proliferate across the network in the case of a distributed system. To address this problem of allowing resource providers to publish resources and for resource users to find these resources in an efficient and inexpensive manner requires the resolution of the following *forces*:

- *Availability.* A resource user should be able to find out on demand what resources are available in its environment.
- *Bootstrapping.* A resource user should be able to obtain an initial reference to a resource provider that offers the resource.
- *Location independence.* A resource user should be able to acquire a resource from a resource provider independent of the location of the resource provider. Similarly, a resource provider should be able to provide resources to resource users without having knowledge of the location of the resource users.
- *Simplicity.* The solution should not burden a resource user when finding resources. Similarly, the solution should not burden a resource provider providing the resources.

Solution Provide a lookup service that allows resource providers to make resources available to resource users. The resource provider advertises resources via the lookup service along with properties that describe the resources that the resource providers provide. Allow resource users to first find the advertised resources using the properties, then retrieve the resources, and finally use the resources.

For resources that need to be acquired before they can be used, the resource providers register references to themselves together with properties that describe the resources that the resource providers provide. Allow resource users to retrieve these registered references

from the resource providers and use them to acquire the available resources from the referenced resource providers.

For resources, such as concurrently reusable services, that do not need to be first acquired and can instead be accessed directly, the resource providers register references to the resources together with properties that describe the resources. Allow resource users to directly access the resources without first interacting with the resource providers.

The lookup service serves as a central point of communication between resource users and resource providers. It allows resource users to access resource providers when resources need to be explicitly acquired from the resource providers. In addition, the lookup service allows resource users to directly access resources that do not need to be acquired from resource providers. In both cases, the resource users need not know about the location of the resource providers. Similarly, the resource providers don't need to know the location of the resource users that want to acquire and access the resources that they provide.

Structure

The following participants form the structure of the Lookup pattern:

- A *resource user* uses a resource.
- A *resource* is an entity such as a service that provides some type of functionality.
- A *resource provider* offers resources and advertises them via the lookup service.
- A *lookup service* provides the capability for resource providers to advertise resources via references to themselves, and for resource users to find these references.

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

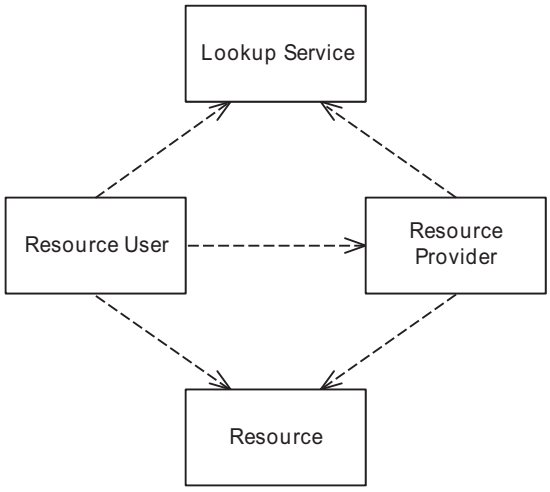
Class **Resource User**	*Collaborator*
Responsibility • Searches for a resource. • Uses a resource.	• Resource • Lookup Service • Resource Provider

Class **Resource**	*Collaborator*
Responsibility • Provides some type of functionality.	

Class **Resource Provider**	*Collaborator*
Responsibility • Provides resources to resource users. • Advertises resources together with their properties via the lookup service.	• Resource • Lookup Service

Class **Lookup Service**	*Collaborator*
Responsibility • Allows resource providers to advertise resources. • Allows resource users to find advertised resources. • Associates properties with resources.	

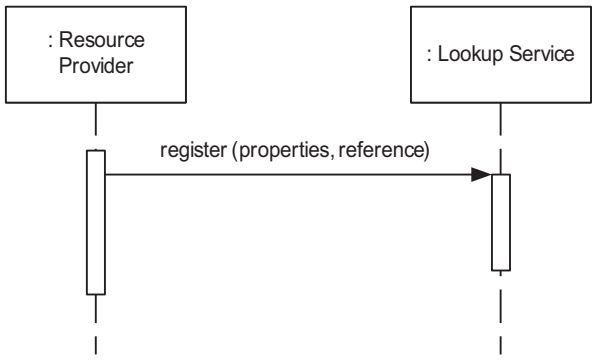
The following class diagram illustrates the structure of the Lookup pattern.



The resource user depends on all three other participants: the lookup service to find the resource provider, the resource provider to acquire the resource, and the resource to actually access it.

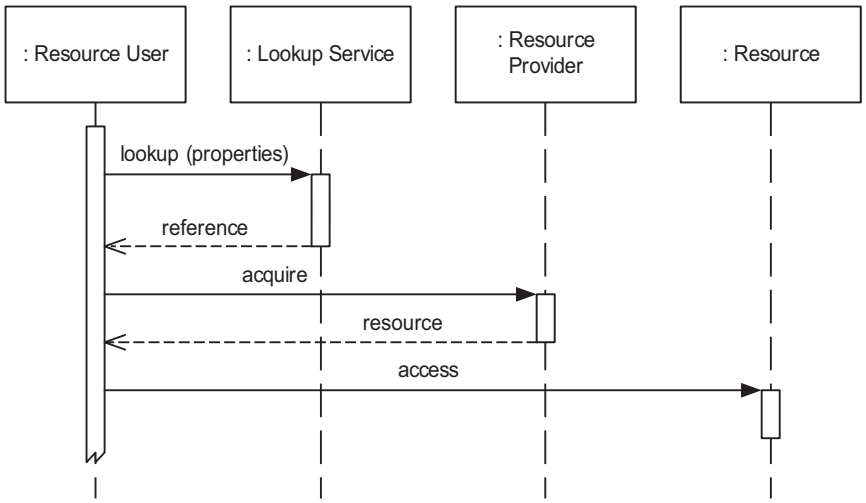
Dynamics There are three sets of interactions in the Lookup pattern.

Scenario I. In the first interaction, a resource provider advertises a resource with the lookup service. It is assumed that the resource provider already knows the access point of the lookup service. For the interactions necessary when the resource provider has no access point to the lookup service, refer to Scenario III. On advertisement of the resource, the resource provider registers a reference to itself with the lookup service, together with some properties that are descriptive of the type of resources that the resource provider provides.



Scenario II. In the second scenario, a resource user finds a resource provider using a lookup service, and includes the following interactions:

- The resource user queries the lookup service for the desired resource using one or more properties, such as resource description, interface type, and location.
- The lookup service responds with the reference to the resource provider, which provides the desired resource.
- The resource user uses the reference of the resource provider to acquire and access the resource.

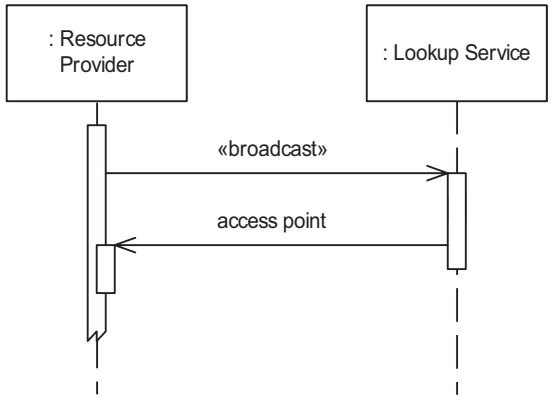


Scenario III. In distributed systems the access point of the lookup service might not be known to the resource user and the resource provider. In such cases the access point might be configured via the runtime environment of the application, or the application might use a bootstrapping protocol to find the access point. The bootstrapping protocol may be a broadcast, multicast or a combination of several unicast messages.

The necessary steps are:

- The resource provider or resource user searches for a lookup service via a bootstrapping protocol.
- The lookup service responds announcing its access point.

The following sequence diagram shows these interactions, which are valid for the resource provider as well as the resource user. In the diagram, the resource provider uses broadcast as a bootstrapping protocol.



Implementation There are four steps involved in implementing the Lookup pattern:

1 *Determine the interface for a lookup service.* A lookup service should facilitate advertisement and lookup of resources, either directly or through their resource providers. It should provide an interface that allows resource providers to register and unregister references. The reference is associated with properties that describe the resource offered. The lookup service keeps a list of the registered references and their associated properties. These properties can be used by the lookup service to select one or more resources or resource providers based on queries sent by the resource user. In the simplest case, the properties may just contain the name or type of a single resource that the resource provider provides. Queries from resource users on the lookup service return either a valid reference, or an error code when no matching resource could be found.

Different policies can be defined for the lookup service. For example, the lookup service may support bindings with duplicate names or properties. The lookup service should also provide an interface that allows resource users to retrieve a list of all available resource providers. The search criteria used by the resource users can be a simple query-by-name, or a more complex query mechanism as described in implementation step 5, *Determine a query language*.

2 *Determine whether to register resource references or resource provider references.* Depending on the kind of resource and its acquisition strategy, either the reference to the resource provider or the reference

to the resource is registered with the lookup service. If a resource must first be acquired explicitly by a resource user, then the reference of the resource provider providing that resource should be registered with the lookup service, together with properties that describe the resource provided by the resource provider. It may be advantageous to require explicit acquisition of a resource to control the type of resource user that can access the resource.

When a resource provider provides multiple resources, the reference of a resource provider can be such that it identifies the resource as well. This can be useful in enabling the resource provider to associate acquisition requests with desired resources. See the *Multi-faceted Resource Provider* variant for details.

On the other hand, if resources need not be explicitly acquired and can be directly accessed by resource users, then references to the resources along with properties describing them can be registered with the lookup service. For example, in the case of concurrently reusable resources, resources can be made available directly to resource users by registering references to the resources with the lookup service. Examples of such resources include read-only objects and Web Services [W3C04]. Such resources either do not have any synchronization issues, or can synchronize access themselves.

3 *Implement the lookup service.* Internally, the lookup service can be implemented in many different ways. For example, the lookup service may keep the registered references and their meta information in some kind of a tree data structure, helpful when complex dependencies must be modelled, or in a simple hash map. For non-critical and heavily changing resource advertisements, the information may be stored transiently, while for critical advertisements the associations should be made persistent, with an appropriate backend persistency mechanism.

➥ For example, the CORBA implementation Orbix [Iona04] uses the COS Persistent State Service to persist the name bindings in its Naming Service, which is an implementation of the lookup service. Other CORBA implementations such as TAO [Schm98] [OCI04] persist the bindings using memory-mapped files. ❑

4 *Provide the lookup service access point.* To communicate with the lookup service, an access point is necessary, such as a Java

reference, a C++ pointer or reference, or a distributed object reference, which typically includes information such as the host name and the port number where the lookup service is running. This information can be published to resource providers and resource users by several means, such as writing to a file that can be accessed by resource providers and resource users, or through well-defined environment variables.

➥ For example, a lot of CORBA implementations publish the access point of the Naming Service using property or configuration files, which can be accessed by clients. ❑

If an access point is not published by the lookup service, it will be necessary to design a bootstrapping protocol that can allow resource providers and resource users to obtain the access point. Such a bootstrapping protocol is typically designed using a broadcast or a multicast protocol. The resource provider or user sends an initial request for a reference to a lookup service using the bootstrapping protocol. On receiving the request, typically one or more lookup services send a reply back to the requestor, passing their access points. The resource provider can then contact the lookup service to publish its reference. Similarly, a resource user can contact the lookup services to obtain references to registered resource providers.

➥ In CORBA, a client or server can acquire the access point of a Naming Service using the `resolve_initial_references()` call on the ORB. Internally, the ORB may use a broadcast protocol to acquire the access point of the Naming Service, such as an object reference. ❑

5 *Determine a query language.* The lookup service may optionally support a query language that allows resource users to search for resources using complex queries. For example, a query language could be based on a property sheet that describes the type of resource in which a resource user is interested. When using the lookup service to query a resource, the resource user may submit a list of properties that should be satisfied by the requested resource. The lookup service can then compare the list of properties submitted by the resource user against the properties of the available resources. If a match is found, the reference to the corresponding resource or resource provider is returned to the resource user.

➥ The CORBA Trading Service [OMG04f] is a lookup service that allows properties to be specified corresponding to a resource that is registered with it. Resource providers are server applications that register CORBA objects with it. The object references will point to the server application, but will also identify the CORBA object as a resource. A client, as resource user, can build an arbitrarily complex query using a criterion that is matched against the properties of the registered CORBA objects. The client is returned a reference to the CORBA object in the server application. ❑

Example Resolved

Consider the example in which a client wants to obtain an initial reference to a transaction manager in a distributed CORBA environment. Using the Lookup pattern, a lookup service should be implemented. Most CORBA implementations provide such a lookup service, either in the form of a Naming Service, a Trading Service, or both. These services are accessible via the Internet Inter-ORB Protocol (IIOP) and provide well-defined CORBA interfaces.

In our example, the server that creates a transaction manager is a resource provider. The server should first obtain a reference to the Naming Service, then use it to register the reference of the created transaction manager. The reference contains the access point of the server, and it identifies the registered resource in the server. The C++ code below shows how a server can obtain the reference to the Naming Service and then register the transaction manager with it.

```
// First initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Create a transaction manager
TransactionMgr_Impl *transactionMgrServant =
   new TransactionMgr_Impl;

// Get the CORBA object reference of it.
TransactionMgr_var transactionMgr =
   transactionMgrServant->_this();

// Get reference to the initial naming context
CORBA::Object_var obj =
   orb->resolve_initial_references("NameService");

// Cast the reference from CORBA::Object
CosNaming::NamingContext_var name_service =
   CosNaming::NamingContext::_narrow(obj);
```

```
// Create the name with which the transaction manager will be bound
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Transactions");

// Register transactionMgr object reference in the
// Naming Service at the root context
name_service->bind(name, transactionMgr);
```

Once the transaction manager has been registered with the Naming Service, a client can obtain its object reference from the Naming Service. The C++ code below shows how a client can do this.

```
// First initialize the ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Get reference to the initial naming context
CORBA::Object_var obj =
   orb->resolve_initial_references("NameService");

// Cast the reference from CORBA::Object
CosNaming::NamingContext_var name_service =
   CosNaming::NamingContext::_narrow(obj);

// Create the name with which the transactionMgr
// is bound in the Naming Service
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("Transactions");

// Resolve transactionMgr from the Naming Service
CORBA::Object_var obj = name_service->resolve(name);

// Cast the reference from CORBA::Object
TransactionMgr_var transactionMgr =
   TransactionMgr::_narrow(obj);
```

Once the initial reference to the transaction manager has been obtained by the client, the client can then use it to invoke operations, as well as to obtain references to other CORBA objects and services.

Variants *Self-registering Resources*. The participants, resource provider and resource, can be implemented by the same software artifact. In this case, since there is no distinction between a resource and a resource provider, the reference that is registered with the lookup service will be that of the resource. For example, a resource such as a distributed service can directly register with the lookup service, providing a reference to itself.

Multi-faceted Resource Provider. When a resource provider offers more than one resource that needs to be acquired, the registered reference of the resource provider can also identify the resource. The resource provider registers a unique reference to itself for each resource that it advertises in the lookup service. The reference that is registered with the lookup service would correspond to the resource provider, but would indirectly refer to a unique resource provided by that resource provider.

Resource Registrar. A separate entity can be responsible for registering references with the lookup service other than the one that actually provides the resources. That is, the resource providers need not be the one that register references with the lookup service; this responsibility can be handled by a separate entity called a resource registrar.

Federated Lookup. Several instances of the lookup service can be used together to build a federation of lookup services. The instances of lookup services in a federation cooperate to provide resource users with a wider spectrum of resources. The Half-Object Plus Protocol [Mesz95] pattern describes how to separate lookup service instances while still keeping them synchronized.

A federated lookup service can be configured to forward requests to other lookup services if it cannot fulfill the requests itself. This widens the scope of queries and allows a resource user to gain access to additional resources it was not able to reach before. The lookup services in a federation can be in the same or different location domains.

Replicated Lookup. The lookup service can be used to build fault-tolerant systems. Replication is a well-known concept in providing fault tolerance and can be applied at two levels using lookup service:

- Firstly, the lookup service itself can be replicated. Multiple instances of a lookup service can serve to provide both load balancing and fault tolerance. The Proxy [GoF95] pattern can be used to hide the selection of a lookup service from the client. For example, several ORB implementations provide smart proxies [HoWo03] on the client side that can be used to hide the selection of a particular lookup service from among the replicated instances of all the lookup services.

- Secondly, both the resources and the resource providers whose reference are registered with a lookup service can also be replicated. A lookup service can be extended to support multiple registrations of resource providers for the same list of properties, such as the same name in the case of the CORBA Naming Service. The lookup service can be configured with various strategies [GoF95] to allow dispatch of the appropriate resource provider upon request from a resource user. For example, a lookup service could use a round-robin strategy to alternate between multiple instances of a transaction manager that are registered with it using the same list of properties. This type of replication is used by Borland [Borl04] to extend the scalability of their CORBA implementation, Visibroker.

Consequences There are several **benefits** of using the Lookup pattern:

- *Availability.* Using the lookup service, a resource user can find out on demand what resources are available in its environment. Note that a resource or its corresponding resource provider may no longer be available, but its reference may not have been removed from the lookup service. See the *Dangling References* liability for further details.
- *Bootstrapping.* The lookup service allows a resource user to obtain initial resources. In distributed systems a bootstrapping protocol allows the resource user to find the lookup service and then use it to find other distributed services.
- *Location independence.* The lookup service provides location transparency by shielding the location of the resource providers from the resource users. Similarly, the pattern shields the location of the resource users from the resource providers.
- *Configuration simplicity.* Distributed systems based on a lookup service need little or no manual configuration. No files need to be shared or transferred in order to distribute, find and access remote objects. The use of a bootstrapping protocol is a key feature for ad hoc networking scenarios, in which the environment changes regularly and cannot be predetermined.
- *Property-based selection.* Resources can be chosen based on properties. This allows for fine-grained matching of user needs with resource advertisements.

There are some **liabilities** of using the Lookup pattern:

- *Single point of failure*. One consequence of the Lookup pattern is the danger of a single point of failure. If an instance of a lookup service crashes, the system can lose the registered references along with the associated properties. Once the lookup service is restarted, the resource providers would need to re-register the resources with it unless the lookup service has persistent state. This can be both tedious and error-prone, since it requires resource providers to detect that the lookup service has crashed and then restarted. In addition, a lookup service can also act as a bottleneck, affecting system performance. A better solution, therefore, is to introduce replication of the lookup service, as discussed in the *Variants* section.
- *Dangling references*. Another consequence of the Lookup pattern is the danger of dangling references. The registered references in the lookup service can become outdated as a result of the corresponding resource providers being terminated or moved. In this case the Leasing (149) pattern, as applied in Jini [Sun04c], can help, by forcing the resource providers to prolong the 'lease' on their references regularly if they do not want their entries removed automatically.
- *Unwanted replication*. Problems can occur when similar resources with the same properties are advertised but replication is not wanted. Depending on the implementation of the lookup service, multiple instances of the same resource may be registered erroneously, or one resource provider may overwrite the registration of a previous resource provider. Enforcing that at least one of the properties is unique can avoid this problem.

Known Uses

CORBA [OMG04a]. The Common Object Services Naming Service and Trading Service implements lookup services. Whereas the query language of the Naming Service is quite simple, using just names, the query language of the Trading Service is powerful and supports complex queries for components.

LDAP [HoSm97]. The Lightweight Directory Access Protocol (LDAP), defines a network protocol and information model for accessing information directories. An LDAP server allows the storage of almost any kind of information in the form of text, binary data, public key

certificates, URLs, and references. Often LDAP servers are protected by permissions, as they contain critical information that must be secured from unauthorized access. LDAP clients can query the information stored on LDAP servers. Large organizations typically centralize their user databases of e-mail, Web and file sharing servers using LDAP directories.

JNDI [Sun04f]. The Java Naming and Directory Interface (JNDI) is an interface in Java that provides naming and directory functionality to applications. Using JNDI, Java applications can store and retrieve named Java objects of any type. In addition, JNDI provides querying functionality by allowing resource users to look up Java objects using their attributes. Using JNDI also allows integration with existing naming and directory services such as the CORBA Naming Service, RMI registry [Sun04g], and LDAP.

Jini [Sun04c]. Jini supports ad hoc networking by allowing services to join a network without requiring any pre-planning, installation, or human intervention, and by allowing users to discover devices on the network. Jini services are registered with Jini's lookup service, and these services are accessed by users using Jini's discovery protocol. To increase network reliability, the Jini lookup service regularly broadcasts its availability to potential clients.

COM+ [Ewal01]. The Windows Registry is a lookup service that allows resource users to retrieve registered components based on keys. A key can be either a ProgId, a GUID (Global Unique IDentifier) or the name and version of a component. The registry allows then to retrieve the associated components.

UDDI. The Universal Description, Discovery, and Integration protocol (UDDI) [UDDI04] is a key building block of Web Services [W3C04]. The UDDI allows publishers of Web Services to advertise their service and clients to search for a matching Web Service. The advertisements contain service descriptions and point to detailed technical specifications that define the interfaces to the services.

Peer-to-peer networks. Peer-to-Peer (P2P) networking technologies, such as JXTA [JXTA04], support advertisement and discovery of peers and resources. Resources in the context of P2P are often files and services

DNS [Tane02]. The Domain Name Service (DNS) is responsible for the coordination and mapping of domain names to and from IP addresses. It consists of a hierarchy of name servers that host the mapping. It is therefore a good example of how a federated lookup works. Clients query any nearby name server, sending a UDP packet containing a name. If the nearby name server can resolve it, it returns the IP address. If not, it queries the next name server in the hierarchy using a well-defined protocol.

Grid computing [BBL02] [Grid04] [JAP02]. Grid computing is about the sharing and aggregation of distributed resources such as processing time, storage, and information. A grid consists of multiple computers linked to form one virtual system. Grid computing uses the Lookup pattern to find distributed resources. Depending on the project, the role of the lookup service in the system is often referred to as 'resource broker', 'resource manager', or 'discovery service'.

Eclipse plug-in registry [IBM04b]. Eclipse is an open, extensible IDE that provides a universal tool platform. Its extensibility is based on a plug-in architecture. Eclipse includes a plug-in registry that implements the Lookup pattern. The plug-in registry holds a list of all discovered plug-ins, extension points, and extensions and allows clients to locate these by their identity. The plug-in registry itself can be found by clients through several framework classes.

Telephone directory service. The Lookup pattern has a real-world known use case in the form of telephone directory services. A person X may want to obtain the phone number of person Y. Assuming person Y has registered his/her phone number with a lookup service, in this case a telephone directory service, person X can then call this directory service and obtain the phone number of person Y. The telephone directory service will have a well-known phone number, for example 411 or a Web site [ATT04], thus allowing person X to contact it.

Receptionist [TwAl83]. A receptionist at a company can be considered as a real-life example of the Lookup pattern. The receptionist manages the contact information in the form of the phone numbers of all the employees of the company. When a caller wishes to contact an employee, they first speak to the receptionist, who then provides the 'reference' of the person being sought. Similarly, if a new person joins the company or an existing employee

leaves, their contact information is typically updated by the receptionist to handle future queries.

See Also

The Activator [Stal00] pattern registers activated components with a lookup service to provide resource users with access to them. In many cases the references retrieved from a lookup service are actually references to factories, implementing the Abstract Factory [GoF95] pattern. This decouples the location of components from their activation.

The Sponsor-Selector [Wall97] pattern decouples the responsibilities of resource selection from resource recommendation and hands these responsibilities to two participants, the selector and the sponsor respectively. Sponsor-Selector can be used in Lookup to improve the finding of a resource that matches the resource user's demand. The role of the sponsor would coincide with the resource provider, while the lookup service would be the selector.

The Service Locator pattern [ACM01] encapsulates the complexity of JNDI lookups for EJB [Sun04b] home objects and the creation of business objects.

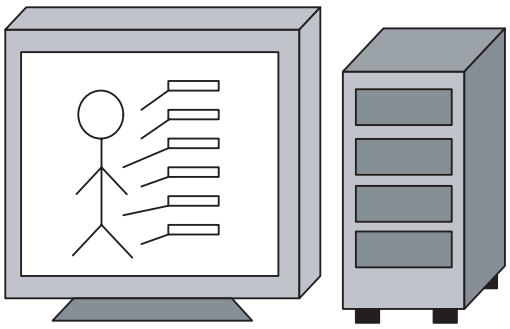
Credits

We would like to thank our EuroPLoP 2000 shepherd, Bob Hanmer, for his feedback and valuable comments. We would also like to thank everyone at the writer's workshop at St. Martin, Austria during our Siemens retreat 2000, Frank Buschmann, Karl Pröse, Douglas C. Schmidt, Dietmar Schütz, and Christa Schwanninger, as well as the people of the writer's workshop at EuroPLoP 2000, Alexandre Duret-Lutz, Peter Gassmann, Thierry Geraud, Matthias Jung, Pavel Hruby, Nuno Meira, James Noble, and Charles Weir, for their valuable comments and suggestions.

Lazy Acquisition

The *Lazy Acquisition* pattern defers resource acquisitions to the latest possible time during system execution in order to optimize resource use.

Example Consider a medical Picture Archiving and Communication System (PACS) that provides storage of patient data. The data includes both patient details such as address information as well as medical history. In addition, the data can include digitized images originating from various sources, such as X-rays and computer tomography scanners. In addition to supporting different sources of patient data, the PACS system must provide efficient access to the data. Such data is typically accessed by physicians and radiologists for diagnosis and treatment purposes.



A PACS system is typically built using a three-tier architecture. The middle tier of the system maintains business objects that represent the patient data. Since the data must be persisted, these business objects and the data they contain are mapped to some persistent store, typically a database. When a physician queries for a particular patient, the data is fetched and the corresponding business object is created. This business object is delivered to the presentation layer of the system, which extracts the relevant information and presents it

to the physician. The time it takes to fetch the relevant patient data and create the business objects is proportional to the size of the patient data. For a patient with a long medical history and several digitized images corresponding to various medical examinations, fetching all the data and creating the corresponding business object can take a lot of time. Since high performance is a typical non-functional requirement of such systems, a delay in fetching patient records can be a big problem.

How can the PACS system be designed so that retrieval of patient information is quick regardless of the number of images in the patient's record?

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

Problem Limited resource availability is a constraint faced by all software systems. In addition, if the available resources are not managed properly, it can lead to bottlenecks in the system and can have a significant impact on system performance and stability. To ensure that resources are available when they are needed, most systems acquire the resources at start-up time. However, early acquisition of resources can result in high acquisition overheads, and can also lead to wastage of resources, especially if the resources are not needed immediately.

Systems that have to acquire and manage expensive resources need a way of reducing 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.

To address these problems requires resolution of the following *forces*:

- *Availability.* Acquisition of resources should be controlled such that it minimizes the possibility of resource shortage and ensures that a sufficient number of resources are available when needed.
- *Stability.* Resource shortage can lead to system instability, and therefore resources should be acquired in a way that has minimum impact on the stability of the system.
- *Quick system start-up.* Acquisition of resources at system start-up should be done in a way that optimizes the system start-up time.

- *Transparency*. The solution should be transparent to the resource user.

Solution Acquire resources at the latest possible time. The resource is not acquired until it becomes unavoidable to do so. When the initial request for a resource is made by the resource user, a resource proxy is created and returned. When the resource user tries to access the resource, the resource proxy acquires the actual resource and then redirects the access request of the resource user to the resource. The resource user is therefore dependent on the resource proxy, but as this provides the same interface as the resource, whether the resource proxy or the resource is accessed is transparent to the resource user.

By using a proxy to represent resources that are potentially expensive to acquire, the overall cost of acquiring a set of resources can be minimized. In addition, by not acquiring a large number of resources up front, the total number of resources that need to be managed simultaneously is also minimized.

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 a connection or memory.
- A *resource proxy* intercepts resource acquisitions by the resource user and hands the lazily acquired resources to the resource user.
- A *resource provider* manages and provides several resources.

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

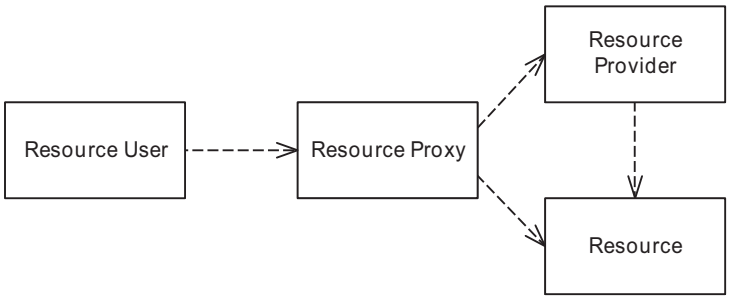
Class **Resource User**	*Collaborator* • Resource Proxy
Responsibility • Acquires and uses a resource.	

Class **Resource**	*Collaborator*
Responsibility • Is acquired and used through the resource proxy.	

Class **Resource Proxy**	*Collaborator* • Resource • Resource Provider
Responsibility • Pretends to be the resource. • Provides the same interface as the resource. • Makes the actual resource available from the resource provider.	

Class **Resource Provider**	*Collaborator* • Resource
Responsibility • Manages and provides resources to resource proxies.	

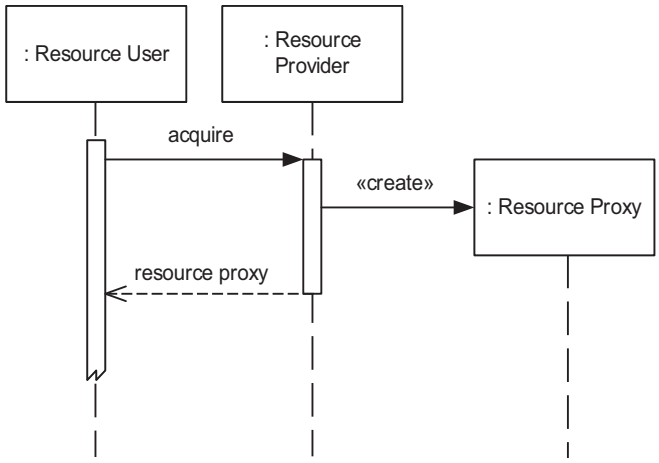
The following class diagram illustrates the structure of the Lazy Acquisition pattern.



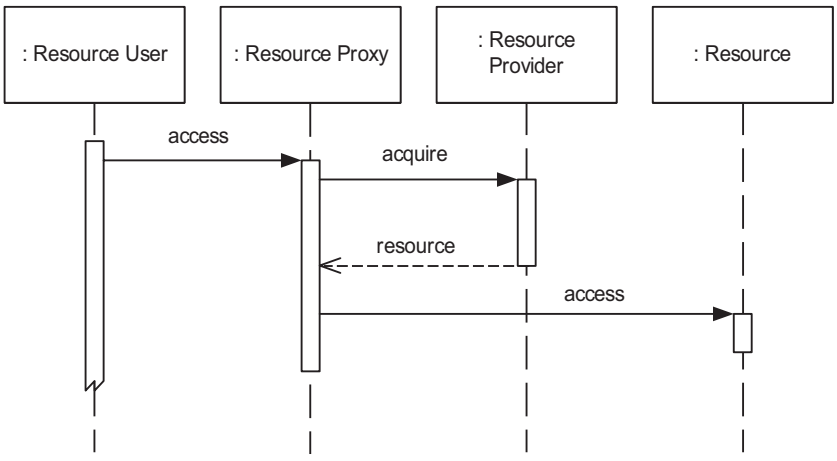
The class diagram shows that the resource user depends on the resource proxy. Since the resource proxy provides the same interface as the resource, whether the resource proxy or the resource is accessed is transparent to the user.

Dynamics **Scenario I.** In this scenario, the resource provider not only provides resources, but also acts as a factory for creating resource proxies.

When the resource user tries to acquire a resource from the resource provider, the resource provider creates and returns a resource proxy to the resource user, instead of the actual resource.



Scenario II. The key dynamics of Lazy Acquisition is the acquisition of the resource by the resource proxy on the first access by the resource user. Initially, the resource proxy does not own the resource. Only at first access is the actual resource acquired. All subsequent access to the resource are forwarded by the resource proxy to the actual resource. The resource user does not notice the level of indirection provided by the resource proxy.



Implementation

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

1 *Identify resources that need to be acquired lazily.* Using profiling and system analysis, identify resources with one or more of the following properties:

- Resources that are expensive to acquire,
- Resources that are available only in limited number, and
- Resources that remain unused for a long time after acquisition.

Review each identified resource and its use, and decide whether through the lazy acquisition of the resource overall resource availability, system stability, and system start-up can be improved. Apply the following implementation steps to each of the identified resources.

2 *Define the resource proxy interface.* For every resource that needs to be acquired lazily, define a resource proxy as Virtual Proxy [GoF95] [POSA1] whose interface is identical to that of the resource.

```
➥  interface Resource {
      public void method1 ();
      public void method2 ();
   }

   public class ResourceProxy implements Resource {
      public void method1 () {
          // ...
      }
      public void method2 () {
          // ...
      }
   }
```

❑

3 *Implement the resource proxy.* Implement the resource proxy such that it hides the lazy acquisition of the resource. The resource proxy will acquire the resource only when the resource is actually accessed by the resource user. Once the actual resource has been acquired, the resource proxy should use delegation to handle all resource access requests. Depending upon the resource, the resource proxy may also be responsible for initializing the resource after acquiring it.

```
➥  public class ResourceProxy implements Resource {
      public void method1 () {
          if (!acquired)
              acquireResource ();
          resource.method1 ();
      }
      public void method2 () {
          if (!acquired)
              acquireResource ();
          resource.method2 ();
      }
      private void acquireResource () {
          // ...
          acquired = true;
      }
      private boolean acquired = false;
   }                                                          ❑
```

4 *Define the acquisition strategy.* Define the strategy by which the resource is actually obtained from the resource provider by the resource proxy. The Strategy [GoF95] pattern can be applied to configure different types of strategy. A simple strategy could be to delay acquisition of the resource until the resource is accessed by the resource user. An alternative strategy could be to acquire the resource based on some state machine. For example, the instantiation of a component might trigger the resource proxy of another existing component to acquire its corresponding resources. In addition, the resource proxy may also provide the ability to switch off lazy acquisition entirely, in which case the resource will be acquired immediately.

➥ The `acquireResource()` method in the code sample below can be used by resource users to trigger the explicit acquisition of the resource by the resource proxy.

```
public class ResourceProxy implements Resource {
   // ...
   void acquireResource () {
       resource = new ResourceX ();
   }
   ResourceX resource;
}                                                             ❑
```

Configure the resource acquisition strategy in the resource proxy, which acquires the resource.

5 *Implement a proper resource release strategy for the lazily acquired resources.* If the resources are not to be released by the resource user

explicitly, then use either the Evictor (168) pattern or Leasing (149) pattern for automatically releasing the resources. For this, the lazily acquired resource needs either to implement the `EvictionInterface`, or be registered with a lease provider, respectively.

Example Resolved

Consider the example of a medical Picture Archiving and Communication System (PACS). In order to solve the problem of quick retrieval of patient information, use the Lazy Acquisition pattern.

When a request is made to fetch all the information for a particular patient, create a query that does not fetch any image data. For each image in the patient record, create an image proxy in the business object that is returned. The presentation layer will process the business object and present all the information. For all the image proxies that it encounters, it will create links in the presentation that is generated. When an image corresponding to such a link needs to be viewed, it can then be fetched (lazily) from the file system. Images are stored on the file system directly, as storing images with their large amounts of binary data in a database is typically inefficient.

Using this solution optimizes fetching of textual patient data, which is typically not large. This can provide the physician with a good summary of the patient's medical history. If the physician wishes to view any image from the patient's record, that can be fetched lazily and on demand.

The sample code below shows the class `PatientManager`, which queries the database for the patient record. It fetches all data except the images.

```
public class PatientManager {
   public static PatientRecord getPatientRecord (String patientId) {
       return dbWrapper.getRecordWithoutImages (patientId);
   }
}
```

The returned `PatientRecord` holds a list of `MedicalExams`, which in turn reference an image each. The class `MedicalExam` cannot differentiate between a regularly loaded image and a lazily loaded image, as the interface to it is the same.

```
interface PatientRecord
{
   String getName ();
   String getAddress ();
```

```
   List getMedicalExams ();
}

interface MedicalExam
{
   Date getDate ();
   // Get the digitized image for this exam
   Image getImage ();
}

interface Image
{
   byte [] getData ();
}
```

The `Image` interface can be implemented using various strategies. The `ImageProxy` class below implements the lazy acquisition of the image by loading it from the file system only when actually accessed.

```
public class ImageProxy implements Image
{
   ImageProxy (FileSystem aFileSystem, int anImageId) {
       fileSystem = aFileSystem;
       imageId = anImageId;
   }

   public byte [] getData () {
       if (data == null) {
           // Fetch the image lazily using the stored ID
           data = fileSystem.getImage (imageId);
       }
       return data;
   }

   byte data[];
   FileSystem fileSystem;
   int imageId;
}
```

Note that the `ImageProxy` needs to have information about how to retrieve the lazily-acquired image from the file system.

Specializations Some specialized patterns derived from Lazy Acquisition are:

Lazy Instantiation [BiWa04b]. 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, lazy instantiation saves cost up front, especially for objects that are not accessed. However, using lazy instantiation can incur a dramatic

overhead in situations in which a burst of users concurrently access objects, leading to a high-demand situation.

Lazy Loading. Defer the loading of a shared library until the program elements contained in that library are accessed. The Component Configurator Pattern [POSA2] can be used to implement this. Lazy Loading is often combined with Lazy Instantiation, since objects need to be instantiated when loaded. Lazy Load [Fowl02] describes how to defer loading the state of an object from the database until a client is actually interested in that object.

Lazy State [MoOh97]. Defer the initialization of the state of an object until the state is accessed. Lazy State is often used in situations in which a large volume of state information is accessed rarely. This pattern becomes even more powerful in combination with Flyweight [GoF95], or Memento [GoF95]. In networks of objects, the Lazy Propagator [FeTi97] describes how dependent objects can determine when they are affected by state changes and therefore need to update their state

Lazy Evaluation [Pryc02]. Lazy evaluation means that an expression is not evaluated until the expression's result is needed for the evaluation of another expression. Lazy evaluation of parameters allows functions to be partially evaluated, resulting in higher-order functions that can then be applied to the remaining parameters. Using lazy evaluation can significantly improve the performance of the evaluation, as unnecessary computations are avoided. In programming languages such as Java or C++, evaluation of sub-conditions in a Boolean expression is done using lazy evaluation, in the form of short-circuiting operators such as '`&&`'.

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. Instead of acquiring a resource lazily as late as possible or at the beginning, the resource can also be acquired at three additional times. 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 in which a topology tree of the network needs to be built.

There are three options:

- Build it when the application starts.
  - *For*: the tree is available as soon as the application is initialized.
  - *Against*: slow start-up time.
- Build it when the user requests it.
  - *For*: fast start-up time.
  - *Against*: the user has to wait for the tree to be constructed.
- Build it after the application has started and before the user requests it.
  - *For*: fast start-up time and tree is available when needed.

The last option is commonly used in network management systems.

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

- *Availability.* Using Lazy Acquisition ensures that not all resources are acquired up front. This helps to minimize the possibility that the system will run short of resources, or will acquire resources when they are not needed.
- *Stability.* Using Lazy Acquisition ensures that resources are acquired only when needed. This avoids needless acquisition of resources up front, thus reducing the possibility of resource exhaustion and making the system more stable.
- *Optimal system start-up.* Using Lazy Acquisition ensures that resources that are not needed immediately are acquired at a later stage. This helps to optimize system start-up time.
- *Transparency.* Using Lazy Acquisition is transparent to the resource user. The resource proxy hides the actual acquisition of the resource from the resource user.

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

- *Space overhead.* The pattern incurs a slight space overhead, as additional memory is required for proxies resulting from the indirection.
- *Time overhead.* The execution of the lazy acquisitions can introduce a significant time delay when a resource is acquired, and also overhead during regular program execution, due to the additional level of indirection. For real-time systems such behavior might not be acceptable.
- *Predictability.* The behavior of a lazy acquisition system can become unpredictable. If multiple parts of a system defer resource acquisition as late as possible, it can lead to bursts when all parts of the system attempt to acquire resources at the same time.

Known Uses

Singleton. Singletons [GoF95], objects that exist uniquely in a system, are usually instantiated using lazy instantiation. In some cases Singletons are accessed by several threads. The Double-Checked Locking [POSA2] idiom can be used to avoid race conditions between threads during instantiation.

Haskell [Thom99]. The Haskell language, like other functional programming languages, allows lazy evaluation of expressions. Haskell only evaluates as much of a program as is required to get the answer. Using this demand-driven evaluation, data structures in Haskell are evaluated just enough to deliver the answer, and parts of them may not be evaluated at all.

Java 2 platform, Enterprise Edition (J2EE) [Sun04b]. Enterprise JavaBeans (EJB) containers in J2EE application servers [Iona04] host many different components simultaneously. To avoid resource exhaustion, they need to ensure that only components that are actually used by clients are active, while others should be inactive. A typical solution is the application of Lazy Loading and Lazy Instantiation for the components and their state. This saves valuable resources and assures scalability. Also, Java Server Pages (JSPs) are typically compiled into servlets by many J2EE application servers only when they are actually accessed, rather than when they are deployed.

Ad hoc networking [Sun04c] [UPnP04]. In ad hoc networking only temporal relationships between devices and their components exist, so that it becomes too expensive to hold on to resources that are not actually needed currently. This means that components need to be loaded, instantiated, destroyed, and unloaded regularly. Ad hoc networking frameworks therefore 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 [IrDA04].

Operating systems. A common behavior of operating systems is to defer the complete loading of application libraries until they are actually needed. For example, on most Unix systems such as Solaris [Sun04h] and Linux, an environment variable called `LD_BIND_NOW` can be used to specify whether or not the shared objects (.so files) should be loaded using a lazy model. Under a lazy loading model, any dependencies that are labeled for lazy loading will be loaded only when explicitly referenced. By taking advantage of a function call's lazy binding, the loading of a dependency is delayed until it is first referenced. This has the additional advantage that objects that are never referenced will never be loaded.

.NET Remoting [Ramm02]. In .NET Remoting so-called 'singleton remote objects' are objects that can be used by multiple clients, but only one instance of the respective object type can exist at the same time in the server. Those singleton remote objects are only instantiated on first access, even though clients might obtain references to them before the first access.

COM+ [Ewal01]. Just-in-Time (JIT) activation is an automatic service provided by COM+ that can help to 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 activation is supported for COM+ and .NET based applications. .NET applications have to set configuration attributes of the .NET `System.EnterpriseServices` package.

JIT compilation. JIT compilation is heavily used in today's 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 that supports this feature is the IBM J9 JVM [IBM02]. The opposite of JIT compilation is ahead-of-time (AOT) compilation.

Java [Sun04a]. JVM implementations optimize Java class loading typically by loading the classes when the code of that class is to be first executed. This behavior is clearly following the Lazy Loading pattern.

Manufacturing [VBW97]. Just-in-Time manufacturing, as used in many industries such as the automobile industry, follows the same pattern. Parts of an assembly are manufactured as they are needed. This saves the cost of fixed storage.

Eclipse plug-in [IBM04b]. Eclipse is a universal tool platform—an open extensible IDE for anything and nothing in particular. Its extensibility is based on a plug-in architecture that allows every user to become a contributor of plug-ins. While the plug-in declarations that determine the visualization of the plug-ins' features are loaded eagerly, the actual logic, which is contained in Java archives (JAR), is loaded lazily on first use of any of the plug-in's functionality.

Heap compression. The research by [GKVI+03] employs a special garbage collector for the Java programming language that allows a heap smaller than the application's footprint to be used, by compressing temporarily unused objects in the heap. When such an object is accessed again, it lazily uncompresses portions of the object, thereby allocating the necessary memory.

FrameMaker [Adob04]. The desktop publishing program FrameMaker does not open and read referenced graphic files until the page containing the graphic is first displayed. The graphic is then rendered on the screen, and the rendered image is stored in temporary storage. When the page is displayed again, Framemaker checks for any updates to the referenced file. If no changes have been made to that file, it reuses the rendered image from temporary storage.

See Also The Eager Acquisition (53) pattern can be conceived as the opposite of Lazy Acquisition. Eager Acquisition describes the concept of

acquiring resources up front to avoid acquisition overheads at the first access by clients.

Since both Lazy Acquisition and Eager Acquisition can be sub-optimal in some use cases, the Pooling (97) pattern combines both into one pattern to optimize resource usage.

The Lazy Optimization [Auer96] pattern can help to tune performance once the program is running correctly and the system design reflects the best understanding of how the program should be structured.

The Thread-Specific Storage [POSA2] pattern uses a proxy to shield the creation of keys that identify the associated thread-specific object uniquely in each thread's object set. If the proxy does not yet have an associated key when accessed, it asks the key factory to create a new key.

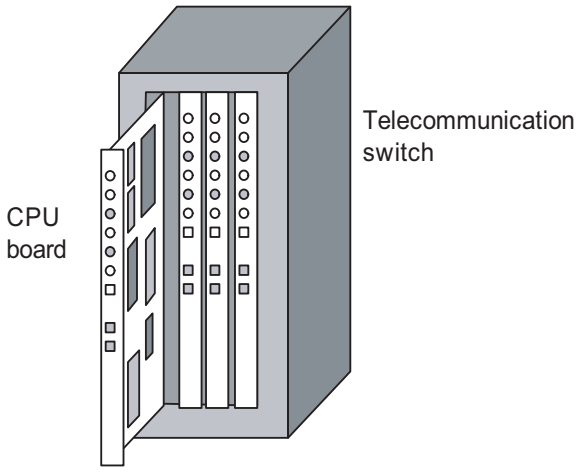
Credits Thanks to Frank Buschmann and Markus Völter for their valuable comments on earlier versions of this pattern. Special thanks to our EuroPLoP 2002 shepherd, Kevlin Henney, and the writer's workshop participants Eduardo Fernandez, Titos Saridakis, Peter Sommerlad, and Egon Wuchner.

Our copy-editor Steve Rickaby provided us with the nice known use for FrameMaker. FrameMaker was used for the whole of the writing and production of this book.

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 use.

Example Consider an embedded telecommunication application with soft real-time constraints, such as predictability and low latency in execution of operations. Assume the application is deployed on a commercial off-the-shelf (COTS) operating system such as Linux, primarily for cost as well as portability reasons.



In most operating systems, operations such as dynamic memory allocation can be very expensive. The time it takes for memory allocations via operations such as `new()` or `malloc()` depends on the implementation of the operations. In most operating systems, including Real-Time Operating Systems (RTOS), the time it takes to execute dynamic memory allocations varies.

The main reasons for this are:

- Memory allocations are protected by synchronization primitives.
- Memory management, such as compaction of smaller memory segments to larger ones, consumes time.

If memory compaction is not done often enough, memory allocations can easily cause memory fragmentation. However, most operating systems, including some RTOS such as VxWorks [Wind04], do not provide such memory management, which can leave an application susceptible to memory fragmentation.

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

Problem Systems with soft real-time constraints need to be stringent about how and when they acquire resources. Examples of such systems include critical industrial systems, highly scalable Web applications, or even the graphical user interface (GUI) of a desktop application. In each case, the users of such systems make certain assumptions about the predictability, latency, and performance of the system. For example, in the case of the GUI of a desktop application, a fast response is expected by the user and any delay in response can be irritating. However, if the execution of any user-initiated request results in expensive resource acquisition, such as dynamic acquisition of threads and memory, it can result in unpredictable time overheads. How can resources be acquired in systems with soft real-time constraints while still fulfilling the constraints?

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 application 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.

Solution Eagerly acquire a number of resources before their actual use. At a time before resource use, optimally at start-up, the resources are acquired from the resource provider by a provider proxy. The resources are then kept in an efficient container. Requests for resource acquisition from resource users are intercepted by the provider proxy, which accesses the container and returns the requested resource.

The time at which the resources are acquired can be configured using different strategies. These 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 start-up, or at a dedicated, possibly calculated, time after system start-up. 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 *provider proxy* intercepts resource acquisitions by the user and hands the eagerly-acquired resources to the resource user in constant time.
- A *resource provider* provides and manages several resources.

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

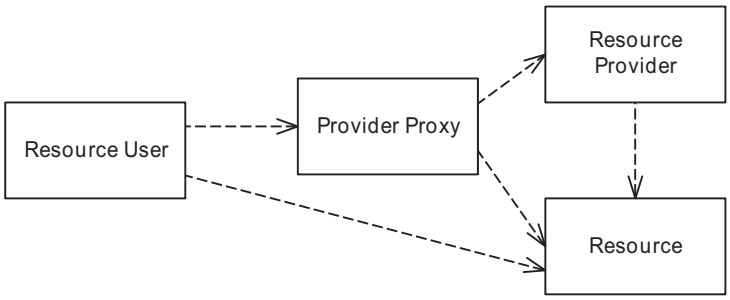
Class **Resource User**	*Collaborator*
Responsibility • Acquires and uses resources.	• Resource • Provider Proxy

Class **Resource**	*Collaborator*
Responsibility • Is acquired from the resource provider by the provider proxy and used by the resource user.	

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

Class **Resource Provider**	*Collaborator*
Responsibility • Manages and provides resources to resource users.	• Resource

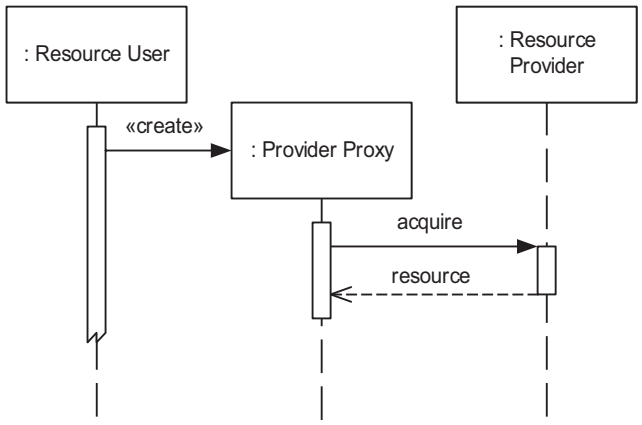
The following class diagram visualizes the structure of the Eager Acquisition pattern.



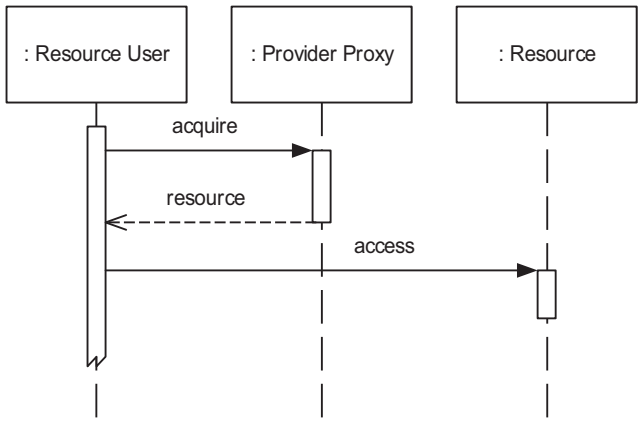
The resource user does not acquire the resource directly from the resource provider, but acquires it via the provider proxy.

Dynamics **Scenario I.** The resource user creates a provider proxy, which it later uses to acquire the resource. The resource is acquired by the provider

proxy before its use. At the latest, the resource is acquired when the resource user actually tries to acquire it.



Scenario II. The resource user acquires a resource, but is intercepted by the provider proxy. The following sequence diagram shows how the resource user acquires the resource.



The provider proxy intercepts the acquisition request and returns an eagerly-acquired resource. The resource user can now access and use the resource.

Implementation Five steps are involved in implementing the Eager Acquisition pattern.

1 *Select resources to acquire eagerly.* Determine the kind of resources to be eagerly acquired in order to guarantee predictable behavior of the overall system. Determine resources that are expensive, such as connections, memory, and threads. The acquisition of such resources is most likely to introduce unpredictability, and therefore they serve as ideal candidates for eager acquisition.

2 *Estimate resource usage.* Estimate the amount of resources a resource user will acquire during its lifetime. Perform test runs and measure the maximum resource usage, if it is not known up front. If you cannot predict the resource use, make it a configuration parameter, so it can be tuned easily after deployment. Provide operator information, such as log entries, for the actual resource utilization of eagerly acquired resources.

3 *Implement the provider proxy.* A provider proxy is responsible for the transparent integration of the Eager Acquisition pattern. It can be included in an actual design in several ways, for example by using a Virtual Proxy [GoF95] [POSA1] or Interceptor [POSA2]. A provider proxy can be obtained from an Abstract Factory [GoF95] by the resource user. When transparency is not required, the resource user acquires the resource directly from the resource provider, in which case a provider proxy is not required.

4 *Implement a container.* Implement a container, such as a hash map, to hold the eagerly-acquired resources. The container should allow predictable lookups. For example, a hash map can provide lookup in constant time.

5 *Determine a timing strategy.* Decide on a strategy, depending on when the provider proxy eagerly acquires 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, although it must be possible to predict the resource usage.
- *Proactively during run time.* Use Reflection [POSA1] to detect system state that might lead 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. However, the cost of proactive behavior is higher complexity and execution overhead for continuous system monitoring.

6 *Determine initialization semantics.* Decide on how to initialize acquired resources to avoid initialization overhead. For some resources, a complete initialization on acquisition by the provider proxy is impossible. In such cases the initialization overhead during run time should be taken into account.

Example Resolved

Consider the embedded telecommunication application described earlier. To make the application predictable, three options exist:

- Implement the application objects as global variables, which is basically eager instantiation of objects.
- Put the objects on the stack, avoiding dynamic memory allocation altogether.
- Implement a memory pool that eagerly acquires memory from the operating system up front, after system start-up but before the first time dynamic memory allocation is needed. The memory pool is used by application objects to acquire their memory.

The first option has the disadvantage of losing control over initialization and release of memory. Further, many developers consider this design to be a poor one because of the high coupling it introduces into a system. Class static variables allow the location of definition to be more visible, but still have some of the same coupling problems, as well as issues regarding timing of initialization and release of memory. Using class static variables also hard-wires the number of instances of a class, which reduces the adaptability of a piece of software, as well as its ability to accommodate runtime/load-time variability.

The second option, working only with objects allocated on the stack, demands large stack sizes. Additionally, the lifecycle of the objects would have to map the call stack, otherwise objects could go out of scope while still being used.

The third option is much more flexible regarding the initialization of application objects, but has some restrictions. As described earlier, synchronization overhead and compaction of variable-sized memory blocks are the main reasons for poor predictability of memory

acquisition time. The memory pool can only become more predictable than the operating system if it is able to avoid synchronization and management of variable-sized memory allocations.

To avoid synchronization overhead, memory pools need either to be dedicated to a thread, or to internalize thread-specific issues using, for instance, Thread-Specific Storage [POSA2]. For details, see also the Thread-Local Memory Pool [Somm02] pattern.

The following C++ class implements a simple memory pool for fixed-size blocks of memory without synchronization. It is expected to be used only by a single thread. The memory it provides is acquired eagerly in its constructor. For support of multiple block sizes, the memory pool internal management of blocks would need to be extended, or separate memory pools, one for each block size, would need to be instantiated.

```
class Memory_Pool {
public:
   Memory_Pool (std::size_t block_size, std::size_t num_blocks)
   : memory_block_ (::operator new (block_size * num_blocks)),
       block_size_ (block_size),
       num_blocks_ (num_blocks)
   {
       for (std::size_t i = 0; i < num_blocks; i++)
       {
           void *block = static_cast<char*>(memory_block_) +
               i*block_size;
           free_list_.push_back (block);
       }
   }

   void *acquire (size_t size)
   {
       if (size > block_size_ || free_list_.empty())
       {
           // if attempts are made to acquire blocks larger
           // than the supported size, or the pool is exhausted,
           // throw bad_alloc
           throw std::bad_alloc ();
       }
       else
       {
           void *acquired_block = free_list_.front ();
           free_list_.pop_front ();
           return acquired_block;
       }
   }
```

```
    void release (void *block)
    {
        free_list_.push_back (block);
    }


private:
    void *memory_block_;
    std::size_t block_size_;
    std::size_t num_blocks_;
    std::list<void *> free_list_;
};
```

An instance of the `Memory_Pool` class should be created for each thread that is spawned and needs to allocate memory dynamically. The eager acquisition of the memory blocks in the constructor, as well as acquisition of block-size memory in the `acquire()` method, can throw a `bad_alloc` exception.

```
const std::size_t block_size = 1024;
const std::size_t num_blocks = 32;
// Assume My_Struct is a complex data structure
struct My_Struct {
    int member;
    // ...
};

int main (int argc, char *argv[])
{
    try
    {
        Memory_Pool memory_pool (block_size, num_blocks);

        // ...

        My_Struct *my_struct =
            (My_Struct *) memory_pool.acquire (sizeof(My_Struct));

        my_struct->member = 42;
        // ...
    }
    catch (std::bad_alloc &)
    {
        std::cerr << "Error in allocating memory" << std::endl;
        return 1;
    }
    return 0;
}
```

The `acquire()` method uses the memory that has been eagerly acquired in the constructor of `Memory_Pool`. Ideally, the constructor should allocate enough memory initially so that all subsequent requests by `acquire()` can be fulfilled. However, in case there is insufficient memory to service a request by the `acquire()` method, additional memory would need to be acquired from the operating system. Therefore, `acquire()` would need to handle such cases as well. Of course, implementing acquisition only is not sufficient, and we also need to implement a proper disposal method [Henn03].

The above code only works for structures that are really just C-like. For real classes, the memory pool must be better integrated with the `new` and `delete` operators, such as:

```
void *operator new(std::size_t size, Memory_Pool &pool)
{
   return pool.acquire(size);
}
```

This would allow the struct above to be allocated as:

```
My_Struct *my_struct_a = new(memory_pool) My_Struct;
```

The price of optimization in this particular case is that for deletion, the traditional new/delete symmetry needs to be broken. Therefore, destroy the object explicitly and return its memory to the pool explicitly:

```
my_struct->~My_Struct();
memory_pool.release(my_struct);
```

For a more detailed discussion on how to integrate custom memory management techniques, see [Meye98].

Even though the above implementation makes certain assumptions and imposes some restrictions, it has the advantage of increased predictability of dynamic memory acquisitions. Furthermore, memory fragmentation is avoided, by allocating only fixed-size blocks. For alternative implementations, see the memory pool implementations of ACE [Schm02] [Schm03a] and Boost [Boos04].

Specializations

The following are some specializations of the Eager Acquisition pattern:

Eager Instantiation. In this case objects are instantiated eagerly and managed in a container. When the application, as resource user,

requests new objects, new instances can be handed back from the list.

Eager Loading. Eager Loading applies eager acquisition to the loading of libraries, such as shared objects on Unix platforms, or dynamically linked libraries on Win32. The libraries are loaded up front, in contrast to Lazy Acquisition (38).

Variants *Static Allocation*. Static Allocation, which is also known as Fixed Allocation [NoWe00], or Pre-Allocation, applies Eager Acquisition to the 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 Resource Allocation [Cros02]. Resource acquisitions can be made up front based on indications derived from resource usage by reflection techniques instead of purely basing it on estimations.

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

- *Predictability*. The availability of resources is predictable, as requests for resource acquisitions from the user are intercepted and served instantly. Variation in delay in resource acquisition, incurred by the operating system, for example, is avoided.
- *Performance*. As resources are already available when needed, they can be acquired within a short and constant time.
- *Flexibility*. Customization of the resource acquisition strategy can easily be applied. Interception of resource acquisition from the user allows for strategized acquisition of resources by the provider proxy. This is very helpful in avoiding side effects such as memory fragmentation.
- *Transparency*. As the resources are eagerly acquired from the resource provider without requiring any user involvement, the solution is transparent to the user.

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 be associated with a resource user, and therefore need

to be organized. Caching (83) and Pooling (97) 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 to guarantee fairness among resource users and to avoid resource exhaustion.
- *Over-acquisition*. Too many resources might be acquired up front by a subsystem that might not need all of them. This can lead to unnecessary resource exhaustion. However, properly tuned resource acquisition strategies can help to address this problem. Pooling (97) can also be used to keep a limit on the number of resources that are eagerly acquired.
- *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 system. If resources are not eagerly acquired at system start-up, but later, there is still an overhead associated with it.

Known Uses

Ahead-of-time compilation [Hope02] [NewM04] is commonly used by Java virtual machines to avoid compilation overheads during execution.

Pooling (97). Pooling solutions, such as connection or thread pools typically pre-acquire a number of resources, such as network connections, or threads, to serve initial requests quickly.

Application servers [Sun04b]. In general, a servlet container of an application server offers no guarantee about when servlets are loaded or the order in which they are loaded. However, an element called `<load-on-startup>` can be specified for a servlet in the deployment descriptor, causing the container to load that servlet at start-up.

NodeB [Siem03]. In the software of the Siemens UMTS base station ´NodeB' the connections to various system parts are eagerly acquired at system start-up. This avoids unpredictable resource acquisition errors and delays during system run time.

Hamster [EvCa01]. A real-world known use is a hamster. It acquires as many fruits as possible before eating them in its burrow. The hamster stores the food in its cheek pouch. Unfortunately, no

numbers are available regarding its estimations about how much it acquires eagerly.

Eclipse plug-in [IBM04b]. Eclipse is a universal tool platform—an open extensible IDE for anything and nothing in particular. Its extensibility is based on a plug-in architecture that allows every user to become a contributor of plug-ins. While the plug-in declarations that determine the visualization are loaded eagerly, the actual logic, which is contained in Java archives (JAR), is loaded lazily on first use of the any of the plug-in's functionality.

See Also The opposite of Eager Acquisition is Lazy Acquisition (38), which allocates resources just in time, at the moment the resources are actually used.

The Pooling pattern (97) combines the advantages of Eager Acquisition and Lazy Acquisition into one pattern.

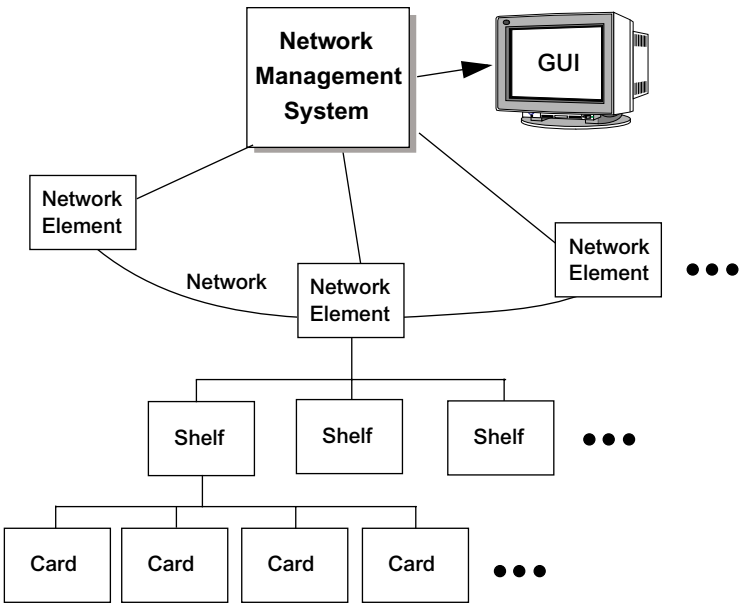
The Caching (83) pattern can be used to manage eagerly-acquired resources.

Credits Thanks to the patterns team at Siemens AG Corporate Technology, to the EuroPLoP 2002 shepherd Alejandra Garrido, and the participants of the writer's workshop, Eduardo Fernandez, Titos Saridakis, Peter Sommerlad, and Egon Wuchner, for their valuable comments and feedback. Further, we are grateful to Andrey Nechypurenko and Kevlin Henney for their suggestions on the example source code of this pattern.

Partial Acquisition

The *Partial Acquisition* pattern describes how to optimize resource management by breaking up acquisition of a resource into multiple stages. Each stage acquires part of the resource, dependent upon system constraints such as available memory and the availability of other resources.

Example Consider a network management system that is responsible for managing several network elements. These network elements are typically represented in a topology tree. A topology tree provides a virtual hierarchical representation of the key elements of the network infrastructure. The network management system allows a user to view such a tree, as well as get details about one or more network elements. Depending on the type of the network element, its details may correspond to a large amount of data. For example, the details of a complex network element may include information about its state as well as the state of its components.



The topology tree is typically constructed at application start-up or when the application is restarting and recovering from a failure. In the first case, the details of all the network elements, along with their components and subcomponents, are usually fetched from the physical network elements. In the latter case, this information can be obtained from a persistent store as well as from the physical network elements. However, in either case obtaining all this information can have a big impact on the time it takes for the application to start up or recover. This is because completely creating or recovering a network element would require creating or recovering all its components. In addition, since each component can in turn be comprised of additional subcomponents, creating or recovering a component would in turn require creating or recovering all its subcomponents. Therefore the size of the resulting hierarchical topology tree, as well as the time it takes to create or recover all its elements, can be hard to predict.

Context Systems that need to acquire resources efficiently. The resources are characterized by either large or unknown size.

Problem Highly robust and scalable systems must acquire resources efficiently. A resource can include local as well as remote resources. Eager acquisition (53) of resources can be essential to satisfy resource availability and accessibility constraints. However, 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. On the other hand, it may not be possible to lazily acquire all the resources, since some of the resources may be required immediately at application start-up or recovery. To address these conflicting requirements of resource acquisition requires resolution of the following *forces*:

- *Availability.* Acquisition of resources should be influenced by parameters such as available system memory, CPU load, and availability of other resources.
- *Flexibility.* The solution should work equally well for resources of fixed size and for resources of unknown or unpredictable size.
- *Scalability.* The solution should be scalable with the size of the resources.

- *Performance*. The acquisition of resources should have a minimal impact on system performance.

Solution Split the acquisition of a resource into two or more stages. In each stage, acquire part of the resource. The amount of resources to acquire at each stage should be configured using one or more strategies. For example, the amount of resources to acquire at each stage can be governed by a strategy that takes into account available buffer space and required response time, as well as availability of dependent resources. Once a resource has been partially acquired, the resource user may start using it on the assumption that there is no need to have the entire resource available before it can be used.

Patterns such as Eager Acquisition (53) and Lazy Acquisition (38) can be used to determine when to execute one or more stages to partially acquire a resource. However, the Partial Acquisition pattern determines in how many stages a resource should be acquired, together with the proportion of the resource that should be acquired in each stage.

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

- A *resource user* acquires and uses resources.
- A *resource* is an entity such as audio/video content. A resource is acquired in multiple stages.
- A *resource provider* manages and provides several resources.

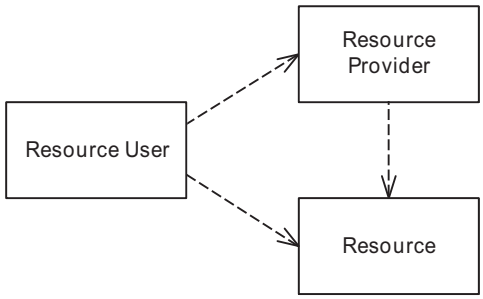
The following CRC cards describe the responsibilities and collaborations of the participants.

Class Resource User	*Collaborator*
Responsibility • Acquires and uses resources.	• Resource

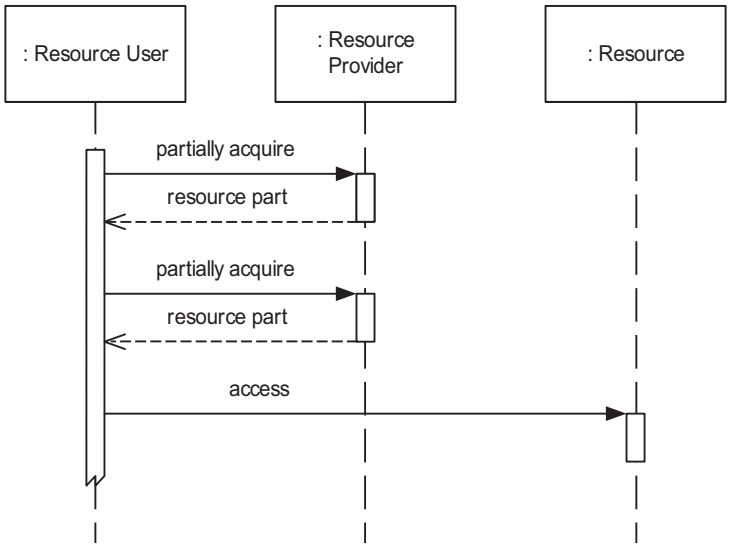
Class Resource	*Collaborator*
Responsibility • Is acquired from the resource provider and used by the resource user.	

Class Resource Provider	*Collaborator*
Responsibility • Manages and provides resources to resource users.	• Resource

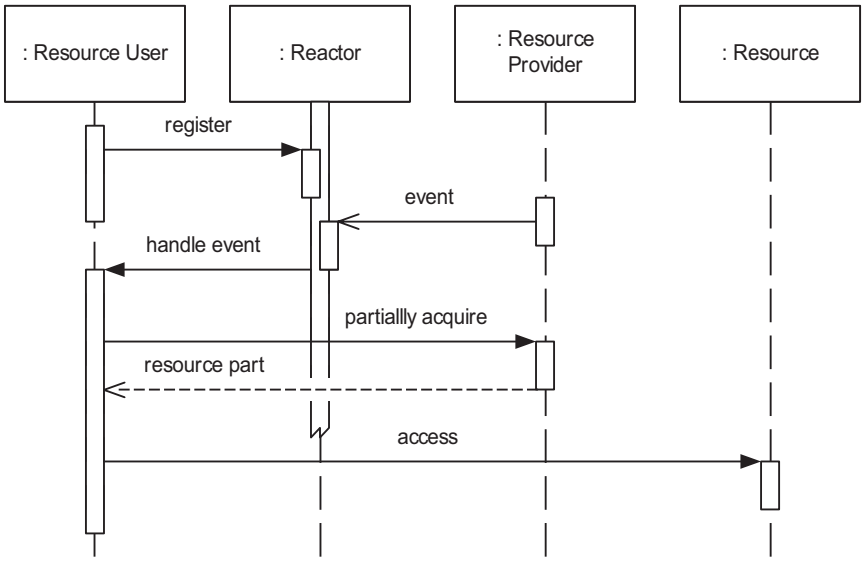
The dependencies between the participants are shown in the following class diagram.



Dynamics **Scenario I.** The sequence diagram below shows how the resource user partially acquires a resource in a series of acquisition steps. When all parts have been acquired, it accesses the resource.



Scenario II. In some cases the resource user might not know when the next part of the resource is ready to be acquired. For example, a resource can be created incrementally, such as the arrival of network packets of a message stream. In such cases the user might not want to block on the acquisition, but be informed about the occurrence of such an event. The Reactor [POSA2] pattern is very useful for such scenarios. Here is how it works dynamically.



The reactor is triggered by some event from the resource provider about the availability of (parts of) the resource. The reactor in turn dispatches this event to the resource user, which in turn performs a partial acquisition. For more details, refer to [POSA2].

Implementation

There are six steps involved in implementing the Partial Acquisition pattern.

1 *Determine the number of stages.* The number of stages in which a resource should be acquired depends on system constraints such as available memory and CPU, as well as other factors such as timing constraints and the availability of dependent resources. For a resource of unknown or unpredictable size it may not be possible to determine the number of stages that it would take to acquire the entire resource. In this case, the number of stages would not have an upper bound, and a new stage would be executed until the resource has been completely acquired.

➥ In the case of the network management system in our example, the number of stages could correspond to the number of hierarchy levels in the topology tree. At each stage, an entire level of the hierarchy can be constructed by obtaining the details of the

components of that level. If a level in the hierarchy is complex and contains a large number of components, obtaining the details of all the components of that level can be further divided into two or more stages. ❑

2 *Select an acquisition strategy.* Determine when each stage of resource acquisition should be executed. Patterns such as Lazy Acquisition (38) and Eager Acquisition (53) can be used to control when one or more stages of resource acquisition should be executed. For example, Eager Acquisition can be used to acquire an initial part of the resource. The remaining parts can then be acquired using Lazy Acquisition, or can be acquired some time in between, after the system has started but before a user requests them.

➥ In the case of the network management system, Eager Acquisition (53) can be used to fetch the details of the network elements, but not of its internal components. The details of the components and sub-components of a network element can be fetched using Lazy Acquisition (38) when the user selects the network element in the GUI and tries to view the details of its components. ❑

3 *Determine how much to acquire.* Configure strategies to determine how much to acquire partially at each stage. Different strategies can be configured to determine how much of a resource should be acquired in each stage. If the size of a resource is deterministic then a simple strategy can evenly distribute the amount of the resource to acquire at each stage among all the stages of resource acquisition. A more complex strategy would take into account available system resources. Thus, for example, if sufficient memory is available, such a strategy would acquire a large part of the resource in a single stage. At a later stage, if system memory is low, the strategy would acquire a smaller part of the resource.

Such an adaptive strategy can also be used if the size of the resource to be acquired is unknown or unpredictable. Additional strategies can be configured that make use of other parameters such as required response time. If there are no system constraints, another strategy could be used to acquire greedily as much of the resource as is available. Such a strategy would ensure that the entire resource is acquired in the shortest possible time. A good understanding of the application semantics is necessary to determine the appropriate strategies that should be used.

4 *Introduce a buffer* (optional). Determine whether the partially-acquired resource should be buffered. Buffering a resource can be useful if the size of the resource is unknown, or if the entire resource needs to be consolidated in one place before being used. If a resource needs to be buffered, the amount of buffer space to allocate should be determined to ensure the entire resource can fit. For a resource of unknown or unpredictable size, a buffer size should be allocated that is within the system constraints (such as available memory) but sufficiently large to handle the upper bound on the size of the resources in the system.

5 *Implement an acquisition trigger.* Set up a mechanism that is responsible for executing each stage of resource acquisition. Such a mechanism would then be responsible for acquiring different parts of a resource in multiple stages. Patterns such as Reactor [POSA2] can be used to implement such a mechanism. For example, a reactor can be used to acquire parts of a resource as they become available. An alternative mechanism can be set up that acquires parts of the resource proactively [POSA2].

6 *Handle error conditions and partial failures.* Error conditions and partial failures are characteristic of distributed systems. When using Partial Acquisition, it is possible that an error occurs after one or more stages have completed. As a result, part of a resource may have been acquired, but the attempt to acquire the subsequent parts of the resource would fail. Depending upon the application semantics, such a partial failure may or may not be acceptable. For example, if the resource being acquired in multiple stages is the contents of a file, then a partial failure would make inconsistent the data that has already been acquired successfully.

➥ On the other hand, in the case of the network management system in our example, the failure to obtain the details of one of the subcomponents will not have an impact on the details of the remaining components acquired successfully. A partial failure could still make the details of successfully acquired components available to the user.

One possible way to handle partial failures is to use the Coordinator (111) pattern. This pattern can help to ensure that either all stages of resource acquisition are completed, or none are. ❑

Example Resolved

Consider the example of a network management system that is responsible for managing a network of several network elements. The network elements themselves consist internally of many components, such as their CPU board, the connection switch, and the memory. Loading the details of those components can take a long time. Using the Partial Acquisition pattern, the acquisition of the details of the network elements, along with their components, can be split into multiple stages. In the initial stage only the details of the network elements will be fetched from the physical network and a database. The details of the components of the network elements will not be fetched.

The topology manager of the network management system provides details about the network element and its components to the visualization subsystem, so that they can be displayed. The topology manager retrieves the information from the database or from the physical network elements, depending on the kind of data, static configuration or current operating parameters. The Java code below shows how a `TopologyManager` can defer fetching the details of the components of a network element to a later stage.

```
public class TopologyManager
{
   // Retrieves the details for a specific network element.

   public Details getDetailsForNE (String neId) {
       Details details = new Details();

       // Fetch NE details either from physical network
       // element or from database ...

       // ... but defer fetching details of NE subcomponents
       // until later. Create a request to be inserted into
       // the Active Object message queue by the Scheduler.

       FetchNEComponents request =
           new FetchNEComponents (neId);

       // Now insert the request into the Scheduler so that
       // it can be processed at a later stage.
       scheduler.insert (request);

       return details;
   }
   private Scheduler scheduler;
}
```

The actual retrieval of network element components is done asynchronously by an Active Object [POSA2]. For this, requests are queued with the `Scheduler`, which runs in the active object's thread. The requests will fetch the network element's components when triggered by the `Scheduler`.

```
public class Scheduler implements Runnable {

   public Scheduler (int numThreads, ThreadManager tm) {
       // Spawn off numThreads to be managed by
       // ThreadManager. The threads would dequeue
       // MQRequest objects from the message queue and
       // invoke the call() method on each one
       // ...
   }

   // Insert request into Message Queue
   public void insert (Request request) {
       queue.insert (request);
   }

   public void run() {
       // Run the event loop
       // ...
   }

   private MessageQueue queue;
}
```

The main logic for retrieving the details of the components of the network element is contained in the `FetchNEComponents` class. It implements the `Request` interface, so that it can be scheduled by the `Scheduler`.

```
public interface Request {
   public void call ();
   public boolean canRun ();
}
```

The `Scheduler` will invoke `canRun()` to check if the `Request` can be executed. If it returns `true` it invokes `call()`, else it reschedules the request.

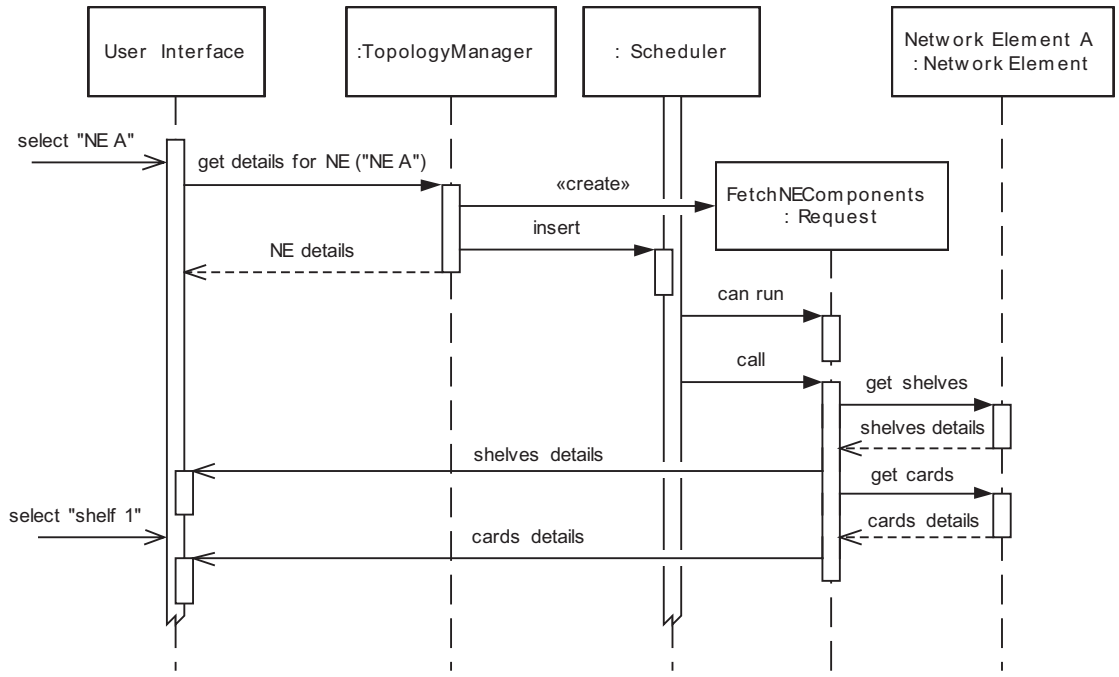
```
public class FetchNEComponents implements Request {
   public FetchNEComponents (String neId) {
       // Cache necessary information
   }

   // Hook method that gets called by the Scheduler.
   public void call () {
```

```
            // Fetch NE subcomponents using Partial Acquisition
        }

        // Hook method that gets called by the Scheduler to
        // determine if this request is ready to be processed.
        public boolean canRun () {
            // ...
            return true;
        }
    }
```

The interactions between the classes are illustrated in the following sequence diagram. When the user selects a network element, the topology manager creates a request that acquires the shelf and card information step by step. At each step the retrieved information is handed to the user interface for visualization. On the selection of a shelf of the same network element by the user, the details are returned quickly, as they have been fetched in the background.



Using Partial Acquisition, the topological tree is therefore constructed in multiple stages. The result is a significant increase in performance when accessing network elements. The first request for details of a

network element, which is usually done at start-up, triggers the partial acquisition of the details of the components of the network element and makes them available by the time the user requests them. In addition, further logic can also be added such that the partial acquisition of the details of a network element can trigger the retrieval of the neighboring network elements. To integrate Partial Acquisition transparently, an Interceptor [POSA2] can be used (see *Variants*).

Variants *Transparent Partial Acquisition.* An interceptor can be introduced that would intercept a resource user's requests for resource acquisition and, using configured strategies, acquire an initial part of a resource. It would then acquire the remaining parts of the resource in additional stages transparent to the resource user. For instance, in the motivating example, an interceptor can be used to intercept requests and only fetch the network elements. The interceptor would not fetch the subcomponents immediately, as they can be fetched at a later stage transparent to the resource user.

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

- *Reactive behavior.* The Partial Acquisition pattern allows acquisition of resources that become available slowly or partially. If this partial acquisition is not done, the resource user would have to wait an undefined time before the resource became completely available.
- *Scalability.* The Partial Acquisition pattern allows the size of the resource being acquired to be scalable. The number of stages in which a resource is acquired can be configured depending upon the size of the resource being acquired.
- *Configurability.* The Partial Acquisition pattern can be configured with one or more strategies to determine in how many stages to acquire a resource, as well as how much of the resource to acquire at each stage.

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

- *Complexity.* User algorithms that handle the resources need to be prepared to handle only partially-acquired resources. This can add a certain level of complexity to applications. In addition, using the Partial Acquisition pattern results in error handling becoming

more complex. If one stage fails, an error-handling strategy must assure that the complete activity is restarted or corrected. On the other hand, error handling also becomes more robust. If a stage acquisition fails, the same stage can be reloaded without the necessity for a complete acquisition restart.

- *Overhead.* The Partial Acquisition pattern requires a resource to be acquired in more than one stage. This can result in an overhead of additional calls being made to acquire different parts of the same resource.

Known Uses

Incremental image loading. Most modern Web browsers such as Netscape [Nets04], Internet Explorer [Micr04], or Mozilla [Mozi04] implement Partial Acquisition by supporting incremental loading of images. The browsers first download the text content of a Web page and at the same time create markers where the images of the page will be displayed. The browsers then download and display the images incrementally, during which the user can read the text content of the page.

Socket input [Stev03]. Reading from a socket also typically makes use of the Partial Acquisition pattern. Since data is typically not completely available at the socket, multiple read operations are performed. Each read operation partially acquires data from the socket and stores it in a buffer. Once all the read operations complete, the buffer contains the final result.

Data-driven protocol-compliant applications. Data-driven applications that adhere to specific protocols also make use of the Partial Acquisition pattern. Typically such applications follow a particular protocol to obtain data in two or more stages. For example, an application handling CORBA IIOP (Internet Inter-ORB Protocol) [OMG04a] requests typically reads the IIOP header in the first step to determine the size of the request body. It then reads the contents of the body in one or more steps. Note that such applications therefore use partially-acquired resources. In the case of an application handling an IIOP request, the application makes uses of the IIOP header obtained in the first stage.

Heap compression. The research by [GKVI+03] employs a special garbage collector for the Java programming language that allows the use of a heap smaller than the application's footprint. The technique

compresses temporarily unused objects in the heap. When a compressed object is accessed again, only a part of the object is uncompressed, resulting in partial allocation of the memory. The remaining parts of the object are uncompressed in subsequent stages, as and when required.

Audio/Video streaming [Aust02]. When decoding audio and video streams the streams are acquired in parts. For videos, the algorithms typically acquire one or several frames at once, which are decoded, buffered, and displayed.

See Also The Reactor [POSA2] pattern is designed to demultiplex and dispatch events efficiently. In the case of I/O operations it allows multiple resources that are acquired partially to be served reactively. Buffering can be used to keep partially-acquired resources in each stage. This can be especially useful if the size of the resource being acquired is unknown.

Credits Thanks to the patterns team at Siemens AG Corporate Technology, to our PLoP 2002 shepherd, Terry Terunobu, and the writer's workshop participants Angelo Corsaro, Joseph K. Cross, Christopher D. Gill, Joseph P. Loyall, Douglas C. Schmidt, and Lonnie R. Welch, for their valuable feedback and comments.