Activator

Authors Michael Stal, Siemens Corporate Technology, Munich, Germany

Douglas C. Schmidt, Vanderbilt University, Nashville, TN, USA

Pattern Description

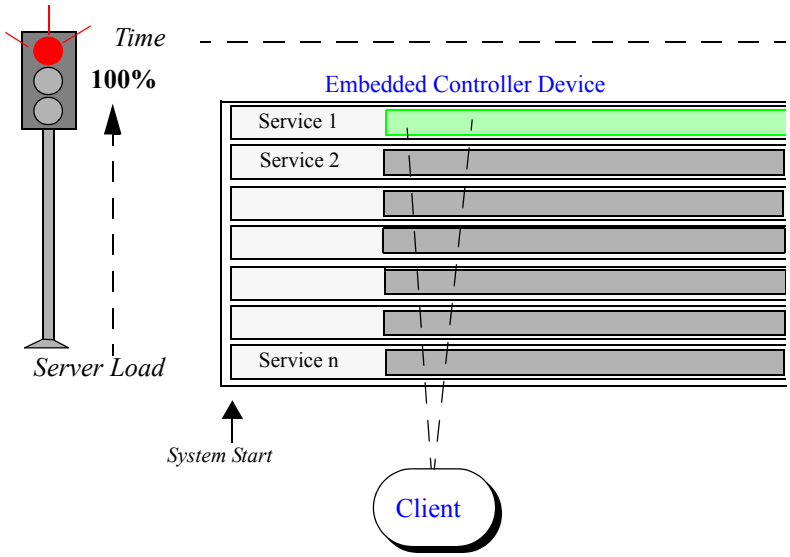
The *Activator* design pattern automates scalable on-demand activation and deactivation of service execution contexts to run services that can be accessed by many clients.

Example

In the industry automation domain, distributed systems such as traffic control systems or manufacturing plants, are increasingly implemented using many embedded devices known as *controllers*. Individual controllers communicate via interconnects, such as CANBUS, Fiberchannel, or Firewire. When software developers build automation systems, they must determine how to provide services, such as inventory trackers, system monitoring, and command and control services, in a manner that scales gracefully as the distribution toplogy and number of clients increases.

In automation systems, service processing must be scalable since multiple clients may access embedded devices concurrently. One service deployment strategy is to apply an *eager resource allocation strategy* [POSA3], which activates processes in controllers during system initialization and runs all services in processes while the system is operational, irrespective of which services are acutally accessed by clients. However, embedded devices often have a limited amount of computing resources, such as main memory, CPU-time, and network connections [SmallMemory]. As the number of clients or services increases, therefore, an eager resource allocation strategy scales poorly because unused server processes consume computing resources that could be allocated more effectively to services actually being accessed by clients.

A typical scenario in the lifetime of an eager resource allocation strategy for a controller in an industrial automation system is shown in the figure below.



In this eager resource allocation scheme, all services are activated automatically at system initialization and consume a considerable amount of available system resources. In the depicted time span above only service 1 is accessed by a client. All other services are busy waiting for incoming requests. The consumption of resources by allocated – but unused – server processes can unnecessarily increase

- **Service response time**, e.g., by competing for resources with services actually accessed by clients, and
- **Hardware costs**, e.g., by requiring more main memory and CPU than would otherwise be needed.

Better service activation strategies are therefore necessary to optimize resource usage and enhance scalability.

Context A resource-constrained distributed computing environment without stringent real-time requirements.

Problem In distributed systems, multiple clients often access services concurrently. These services are deployed in service execution contexts (such as

operating system processes, threads, and/or component containers) and consume limited system resources (such as network/database connections, threads, virtual memory, process table slots, and open files). As a consequence, it is often necessary to balance the following *forces*:

- *Parsimony.* Service execution contexts available in the system should only consume resources for services that are actively accessed by clients.
- *Transparency.* Clients should be shielded from the location, deployment, and management of services.

Solution Minimize resource consumption by activating service execution contexts on demand, running service implementations in these contexts, and deactivating services and their contexts when they are not accessed by clients. Separate service usage from lifecycle aspects to provide location-independent service access.

In detail: Implement *services* that have *service identifiers* and offer functionality to *client* applications via their *service references*. Use *service execution contexts* to manage the lifecycle of these services, in particular their activation, processing, and deactivation. Implement an *activator* to activate service execution contexts on demand and deactivate them again when clients no longer access them. Provide a registration interface that services can use to register and unregister their availability with the activator. Use the service reference to ensure clients only access services via activators. If a service is not running when a client tries to access it, an activator automatically creates the appropriate service execution context and arranges for the service to process the client's request(s) in this context.

Structure A *client* is an application that uses services to perform portions of its computations. It accesses the services remotely using special proxies called *service references* that it obtains from an *activator*.

Class Client	***Collaborator***
Responsibility	• Activator
• Uses services to perform portions of its computation	• Service
• Accesses services via service references	
• Obtains service references from activator	

➥ In our industrial automation system, clients access services within embedded devices by connecting to these devices remotely. Example clients include material flow controllers that identify optimal paths for delivering goods to their destinations and administration consoles that monitor and control an automation system. ❑

A *service identifier* is a token that clients use to identify a particular service. A client passes a service identifier to an activator, which extracts all required information to locate and provide the requested service.

Class Service Identifier	***Collaborator***
Responsibility	• Service Reference
• Identifies a service	• Client
	• Service

➥ In our automation example, the service identifier opaquely encodes a single service's addressing information, including the physical network address of its embedded device, the port address on which an activator listens for incoming requests, and addititional context information, such as session information and security information. ❑

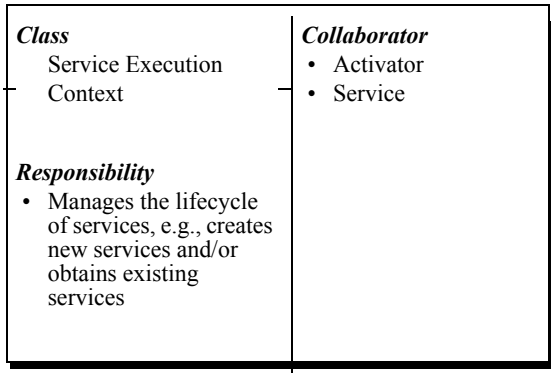
A *service reference* is a proxy [POSA1][GoF] that facilitates client communication with the activator and service. In addition, it shields clients from an activator's involvement in connecting clients and services. A service reference can also encode information about the service and the service execution context that can be used to optimize communication and enhance availability.

Class Service Reference	***Collaborator***
Responsibility • Serves as a proxy to the actual service • Hides activation and deactivation details from clients. • Encodes information about the service and service execution context	• Activator • Service

➥ In our automation example, the service reference is a proxy object that shields the client from system-level details of communication or activation. The service reference uses the service identifier to extract all necessary information to forward client requests to their destinations. ❑

A *service execution context* executes services and controls their activation and deactivation lifecycle. It provides a factory to create services and/or lookup functionality to obtain existing services. Common service execution contexts are operating system processes or threads. Another service execution context is a component container in component

middleware that provides the context for processing operation invocations on components.



➥ Our example uses thread-based service execution contexts to run automation services implemented as C++ objects. After activating a service, the service execution context invokes a method on the service to initialize itself. ❑

A *service* is an entity that is executed in a service execution context and provides functionality or resources to clients. This pattern focuses on services that

- Can be accessed by multiple clients concurrently,
- Require non-trivial utilization of resources, such as memory or processing time,
- Are activated quickly relative to service processing time,
- Are not accessed continuously throughout system lifetime.

Services are named by their service identifiers and accessed by clients via their service references. A service must be registered with an activator manually by users or by some other administrative entity.

Class	***Collaborator***
Service	• Client • Service execution context
Responsibility	
• Provides functionality or resources to clients	

➥ In our automation example, embedded system controllers provide remotely accessible services, such as command and control functionality that allows adminstrators to check and change the current configuration. These service instances run in threads and consume various system resources, such as main memory, CPU time, sockets, or database connections. Multiple clients access these service components at various frequencies, i.e., not all services are accessed all the time. ❑

An *activator* is a mediator [GoF] between services and their clients. It activates service execution contexts on demand. The activator uses an *activation table* to insert and remove registration information about services and their associated service execution contexts. When a client needs to access a currently inactive service, the activator activates a service execution context and arranges for the service to process the upcoming client's request(s) in this context.

A client obtains a service reference from the activator, which it then uses to invoke operations on the service. The activator uses information in its activation table to activate the appropriate service if it is currently inactive. Clients that query the activator for a service must indicate the desired

service via a service identifier, which the activator uses to find the associated entry in its activation table.

Class Activator	***Collaborator*** • Service • Activation Table
Responsibility • Activates and deactivates service execution contexts to run service implementations	

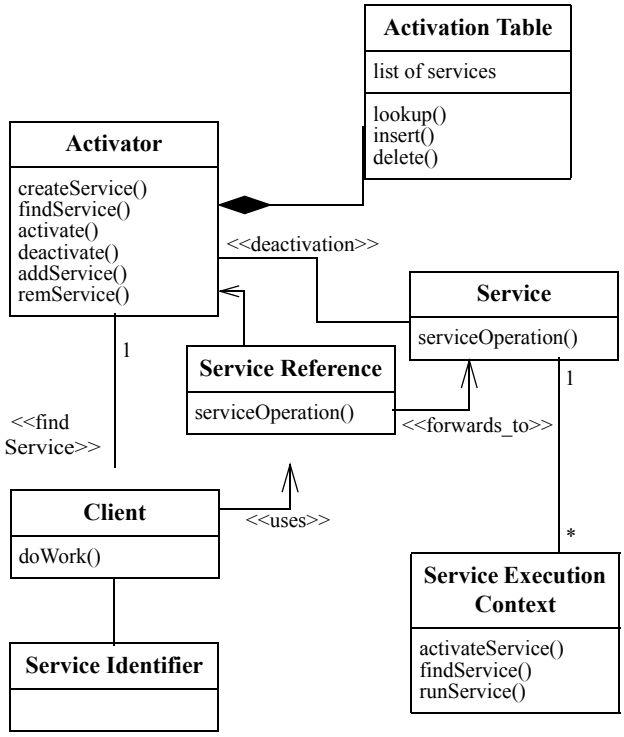
➥ Activation in our automation example can involve different activities. An activator can be implemented as a remote gateway listening on a network port for incoming client requests. A client request is always channeled through a service reference. If the service's execution context has already been created, the activator simply forwards the client request to the service. If the service execution context is not activated, however, the activator creates a thread to execute the service and initializes the service. After this initialization phase, the service reference on the client is associated with the service execution context and the client request is forwarded to the service. ❑

An activator uses its *activation table* to map service identifiers to service implementations and service execution contexts. An activator uses this table to store associated registration and deregistration information when new services become available. These entries may include the execution path of the service executable or DLL, a reference to the service's interface, activation policies, and other configuration information.

➥ The activation table in our automation example is implemented by a hash table that maps service identifiers to associated information, such as the port address of the service execution context, the address of the external service interface, information about the concrete service, a flag indicating whether the service execution context and the service are currently running, and other bookkeeping information. ❑

Class Activation Table	***Collaborator*** • Service
Responsibility • Map service identifiers to service implementations • Manage (i.e., insert, delete, change, and lookup) information on services	

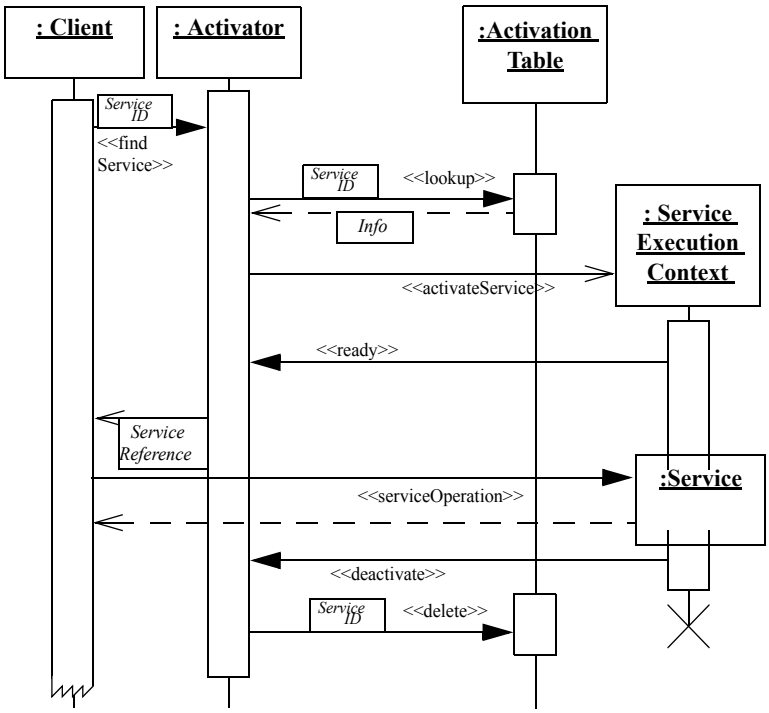
The UML class diagram below illustrates the relationships between the



participants in the Activator design pattern.

Dynamics There are three phases to the dynamics in this pattern: service registration, service access and activation, and service deactivation. The following

figure illustrates these phases, which are described in detail after the figure.



Service registration. The following scenario illustrates how a service is registered with the activator.

- A service developer implements a service using appropriate programming language and platform libraries or middleware.
- The service is registered with the activator. The entity responsible for service registration and the registration time depends on the implementation strategy.

Service access and activation. The following scenario illustrates how a client accesses a service via an activator and then uses the service:

- A client obtains a reference to a service based on the service's identifier.

- The client then accesses a service via its reference, e.g., it invokes an operation on the service.
- The client's request is transparently sent to the activator, which determines the service from the identifier in the request and finds the corresponding entry in the activation table.
- The activator checks whether a service execution context running the service is currently active. If it's inactive, the activator uses activation-related information in its activation table to activate the service execution context that runs the service.
- The activator waits for acknowledgement that the service execution context and the service it implements are activated and ready to receive requests.
- The activator then delegates the request to the service execution context, which carries out the client's request.

Service deactivation. The following scenario illustrates how a service is deactivated.

- When no clients are accessing the service it can be deactivated. How deactivation is triggered is up to the implementation, as discussed in Section 3.3.
- Deactivation may cause the service to store any non-volatile state information in persistent storage and then terminate the service execution context it's running in.

Implementation There are many ways to instantiate the Activator pattern. The following activities focus on the key design and implementation issues, rather than covering all details.

1 *Define clients and the services.* Analyze the application domain for types of services that applications often need.

➥ For example, embedded system controllers typically require services for configuring and monitoring parts of the automation system. These activities represent service types in this application domain. ❑

2 *Identify services that should be activated and deactivated on demand.* For this activity, iterate through the following subactivities:

2.1 *For each service determine the costs for activating and deactivating services, as well as keeping them alive.* The latter costs are measured in terms of resources required by the service types.

➥ For example, although an embedded controller contains a limited amout of computing resources, such as CPU time or memory, monitoring services typically incur high usage of both resources. In contrast, activation time is relatively low, so it makes sense to implement on-demand activation strategies for embedded controllers that don't have hard real-time requirements. ❑

2.2 *Determine client/service usage profiles and identify quality of service (QoS) requirements.* If instances of a particular service are used continuously throughout the whole lifecycle of their clients – and/or if it is critical that clients have low and predictable latency – they may not be good candidates for on-demand activation. An example for such a service might be a real-time controller for an anti-lock braking system, which may not be feasible to activate on demand due to the increased latency and jitter. In contrast, an FTP or SSH login service are often accessed by clients sporatically and don't have stringent latency and predictability requirements, so they are better candidates for on-demand activiation. Another part of the service usage profile is how many instances of a given service must be active at the same time and thus competing for the same resources.

2.3 *Identify services for on-demand activation*. Using the results of the previous subactivities, determine all services that are subject to on-demand activation. As a rule of thumb, such service have the following properties:

- They are used temporarily – not continuously – by clients, so it makes sense to activate and deactivate them on-demand to minimize resource consumption.
- The costs for activating and deactivating these services is negligible compared with the QoS requirements of clients, as well as with the time periods when these services must be available.

➥ In our automation system example, no services have stringent real-time requirements, so they are candidates for on-demand activation via the Activator pattern. ❑

3 *Develop a service activation and deactivation strategy*. For every service, determine the details of service activation and deactivation by performing the following subactivities:

3.1 *Define the service execution context representation*. A common service execution context is an operating system *process*, which provides the unit of virtual memory protection and security, or *thread*, which provides a unit of execution. Another service execution context is a *container*, which provides the runtime context for a service implemented as a component.

➥ Our automation system implements service execution contexts using threads. All initialization information, such as the factory for creating service implementations, is specified declaratively. ❑

3.2 *Define a service initialization strategy*. If all services are stateless, little or no initialization may be required. If they are stateful, however, they must be initialized when they are created. In some cases, the activator or the service execution context can handle initialization issues, e.g., an activator can invoke internal initialization methods of the service based on information stored in its activation table. In some cases, a service may perform its own initialization. In yet other cases, clients may be responsible for initializing their services.

➥ In the automation example, all services are stateless so initialization is simplified and self-contained. ❑

3.3 *Define a service deactivation strategy*. There are several strategies for deactivating services:

- *Service-triggered deactivation*. In this strategy, a service decides to deactivate itself. For example, an activator could choose to deactivate the service if a designated period of time elapses without any clients sending the service requests. This strategy is commonly known as the Evictor pattern [POSA3] [HV99].
- *Client-triggered deactivation*. In this strategy, a client explicitly invokes an operation to trigger deactivation of the service. To implement client-triggered deactivation, the service must be notified whenever a client is obtaining a reference or releasing its reference to this particular service. Internally, the service may keep a reference count that it increments/decrements on service access/release. When the count reaches zero the service is deactivated.

- *Activator-triggered deactivation*. In this strategy, the activator decides when to deactivate a service. For example, the activator might track resource usage on a particular computing node and deactivate services after a certain threshold is crossed. Naturally, care must be taken to deactivate services gracefully to avoid disrupting vital processing and losing important state information.

In most cases, once the service is ready for deactivation it should inform the service execution context so it can release any allocated resources.

➥ "Our automation example uses service-triggered deactivation via the Evictor pattern, i.e., services deactivate themselves and terminate their service execution context if they don't receive any client requests after a certain period of time. ❑

4 *Define the interoperation between services and the service execution context.* The service execution context may provide the following types of operations to its services:

- Operations to access information and resources managed by the execution context,
- Operations to request service deactivation,
- Operations to modify the behavior of the service manager.

Likewise, services might provide

- Global operations for service instantiation,
- Callback methods that the services execution context invokes automatically upon the occurrence of interesting service lifecycle events, such as service activation, deactivation, creation, and destruction.

➥ [Services in the automation example might implement a callback interface the service execution context automatically invokes when a service is activated or deactivated, when it is created, and before it is going to be removed. These callback methods are used to acquire or release resources. ❑

5 *Define the necessary contracts between interoperating participants*. A contract specifies the set of interfaces implemented by each pair of parties that communicate and protocols they must obey. Activity diagrams or interaction diagrams can be used to model the protocol; class diagrams can be used to model the interfaces.

First, determine the following internal contracts that are not visible to clients:

- The contract between the activator and the service execution context specifies how an activator locates, registers, unregisters, and activates a service. It also describes how the activator activates, registers, and unregisters services managed by the service execution context.
- The contract between the service execution context and its services introduces interfaces for creating, initializing, and releasing services. It also specifies how a service can notify its service execution context about its deactivation.

In addition, define the following external contracts that are visible to clients:

- The contract between the client and the activator defines how a client obtains a service reference from the activator. This contract defines a service identifier that encapsulates the primary key necessary for service identification, as well as the primary key for the service execution context where the service implementation runs. An activator knows how to extract these keys from a service identifier.
- The contract between the client and the service defines the set of operations a client can use to access the functionality of the service and how it releases the service after its processing is complete.

Variants

Distributed Activator. In this variant, there is one activator on each network node. When a clients asks for a particular service, that node's activator checks whether the corresponding service is available locally or remotely. In the former case, the workflow continues exactly as in the general pattern. In the latter case, however, the local activator determines on which network node the appropriate service is available and then connects to the remote activator, which retrieves a reference to the service and returns it to the local activator, which in turn returns the service reference to the client. Remote proxies help clients to access remote services in a location-transparent way.

Transparent Activator. In some implementations of the Activator pattern, clients or their service references may be aware that they are retrieving services via an activator. It is often beneficial, however, to shield clients from the activator, so they believe they are accessing the service directly rather than indirectly. To implement the *Transparent Activator* variant an

interceptor, proxy, or mediator agent can be used to transparently contact the activator whenever the service is instantiated.

Multi-Instance Activator. Instead of making the activator a singleton, each service or service execution context could provide its own activator instance. In this variant, the activation table is provided as a global repository accessible by all activator instances. The advantage of this approach is its higher scalability and reliability. However, activator instances must coordinate access to the activation table, which can increase complexity.

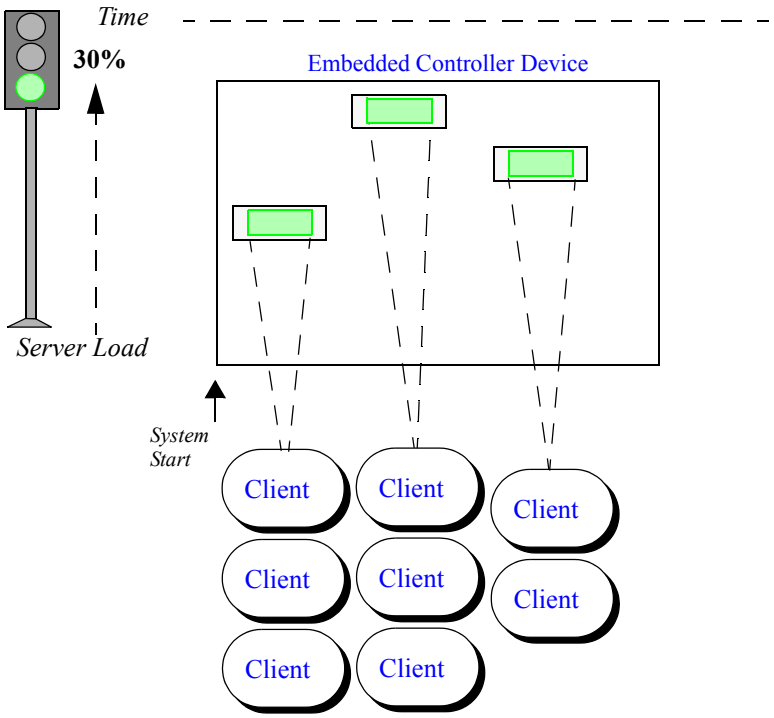
One Service per Service Execution Context. Instead of allowing a service execution context to provide multiple service types, this variant enforces a 1:1 relationship between service execution contents and services. Each service execution context implements exactly one service. The advantage of this approach is a reduced complexity of the activator implementation. However, resource contention increases when more service execution contexts are available. Hence, this approach is primarily advantageous when services have a long execution time or when the number of services is relatively small.

Combined Component Configurator and Activator. This compound pattern combines the Component Configurator pattern [POSA2] with the Activator pattern to provide the ultimate in on-demand flexibility. In this variant, an activator is responsible for activating/deactivating service execution contexts in which services run, whereas a component configurator is responsible for determining what service implementations are actually linked into a server from a dynamic link library (DLL). This compound pattern approach leads to a highly flexible design with well-defined separation of concerns. For example, the activator in such systems could use a component configurator to link and unlink service implementations on-demand from DLLs.

Example Resolved

Applying the Activator pattern to the industrial automation system as described in the *Implementation* section improved the scalability of the system by ensuring that computing resources are consumed only by services being accessed by clients. The diagram below shows that activating services only on demand improves system scalability. In the initial implementation, only a small number of clients could access the system at the same time since limited system resources were devoted to running unused services. In the refactored implementation, a larger

number of clients can access the same or different services concurrently without causing overload.



After refactoring of the initial eager resource allocation strategy, the revised system activates service execution contexts and services on-demand and deactivates them when clients don't access them after a designated period of time. Some addition runtime overhead is caused by the activator spawning threads to run newly activated services, but this overhead is negligible since each client exchanged a number of requests with the service before focusing its attention elsewhere.

Known Uses **Object Request Broker (ORB) and Component Middleware frameworks,** such as CORBA, CORBA Component Model (CCM), Microsoft COM+, and Java RMI use the Activator pattern in several ways:

- They use the pattern to transparently spawn server processes when clients invoke operations on remote objects.

- For example, in COM+ the Service Control Manager (SCM) can spawn server processes on demand. It then connects to the appropriate class factory and creates a new instance of a COM object. The activation table is implemented by a combination of the Windows registry and internal tables. A global DLL, called OLE32.DLL encapsulates access to the activator implementation transparently for clients.
- CORBA ORBs use the *Transparent Activators* variant to activate servers on demand. When a client invokes an operation on an object reference, the call initially goes to an Implementation Repository [VH99], which plays the role of the activator in this pattern. The Implementation Repository checks to see if a server process containing the object being accessed by the client is running. If it's not running, the server process is spawned. After the Implementation Repository verifies the process is running, it returns a `LOCATION_FORWARD` message to the client ORB, which updates the object reference to note the new location and reissues the call to the server transparently to the client application.

- The pattern is used to transparently activate components via a hierarchy of actiavtors. For example, in the CORBA Component Model (CCM) the Implementation Repository is used to spawn server processes. Servant activators can then be used to create containers that provide the runtime environment for managing the lifecycle of component implementations. Similar mechanisms are available in Enterprise JavaBeans.

Network superservers. The Activator pattern has been used in 'superservers' that manage network servers. Two widely available network server management superservers are `Inetd` [Ste90] and `Listen` [Rago93]. Both frameworks consult configuration scripts that specify the following information:

- *Service names*, such as the standard Web and Internet services `HTTP`, `TELNET`, `FTP`, `DAYTIME`, and `ECHO`,
- *Port numbers* to listen on for clients to connect with these services, and
- *An executable file* to invoke and perform the service when a client connects.

Both `Inetd` and `Listen` contain a master acceptor process that monitors a set of port numbers associated with the services. When a client

connection occurs on a monitored port, the acceptor process accepts the connection and demultiplexes the request to the appropriate pre-registered service handler. This handler performs the service, either reactively, proactively, or as an active object [POSA2], and returns results to the client.

Web servers use the Activator pattern to start services on demand when HTTP requests arrive. Plug-ins may be registered with the Web server (e.g., using configuration files or Component Configurators [POSA2]) which represent service execution contexts. These plug-ins are responsible to handle HTTP requests for a specific (namespace of) URL addresse(s). For example, when the URL specifies a file with a PHP file-extension, a PHP-plug-in is accessed by the web server to handle this kind of request. Handling the request in this context means to load the PHP interpreter, execute the PHP-script specified, and return a HTML page to the originator of the request. To optimize performance, the server only activates plug-ins on demand when an appropriate request arrives.

Human Usage. A human known use of the Activator pattern is a call center used to provide technical help desk services, credit card fraud reporting, or airline reservations. Here the resources to be optimized are telephone lines, computer and databased connections, and call center operators. The activator is the central system that is called by customers. After a customer has specified their service indentifier via voice or touchtone input, the call center activator connects the customer to the appropriate operator, after first activating the resources needed by the operator to handle the call, which can involve establishing network and database connections, preparing information on the user interface display, etc. The customer is then connected directly to the operator. Hanging up the telephone triggers service deactivation and releases the allocated resources for use in servicing other customer calls.

Consequences The Activator pattern offers the following **benefits**:

Scalable resource usage. Service execution contexts only run when services are being accessed by clients. They are deactivated and reactivated on demand, which helps improve the scalability of the overall system by allocating resources more effectively.

Implicit initialization. All details of service and service execution context activation and deactivation are encapsulated by the activator interface, which enables service developers to initialize services when they are

activated. For example, service state can be stored in a database and loaded whenever the service execution context is activated., so clients may not need to initialize services explicitly themselves.

Exchangeable strategies due to transparent service creation. As a consequence of using activators as intermediaries, the service creation strategy can be exchanged without impacting clients. For example, an activator can choose between different services supporting the same service type via load balancing or fault tolerance replication mechanisms.

Location transparency with respect to services. If the service references returned by the activator point to proxies, the location of the service can be made invisible to clients. Clients can thus access services residing on remote machines transparently.

Efficient and fast service access. After clients have obtained updated service references from an activator, they can access the services directly, bypassing further indirection and delegation.

On the other hand be aware of the following **liabilities**:

QoS penalties due to activation. When a client first accesses an inactive service, the activator must activate a server execution context to run the service, which increases the latency and jitter of the initial access.

Complex state management. If service execution contexts running services are deactivated and activated on demand, any non-volatile state must be persisted across succeeding passivation and activation events, which can complicate service development.

Debugging and testing can be hard. Decoupling clients from the activation of services can make it harder to determine why failures occur. For example, if there is not enough memory to activate a service in a service execution context, the client may not be able to ascertain what caused the problem since service activation is supposed to be transparent.

See Also The *Component Configurator* design pattern [POSA2] allows applications to dynamically link and unlink their component implementations at run-time without having to modify, recompile, or statically relink application code. The primary difference between Component Configurator and Activator is that Activator focuses on activating/deactivating a service execution context on-demand, whereas Component Configurator focuses on dynamic linking/unlinking the code that runs within an execution

context. The Component Configurator and Activator patterns can be combined into a compound pattern, as described in the *Variants* section.

The *Virtual Component* [PLoP9] and *Virtual Proxy* design patterns [POSA1] can be used in conjunction with the Component Configurator pattern to provide an transparent way of loading and unloading components that implement middleware and/or application software functionality. This pattern ensures that the software provides a rich and configurable set of functionality, yet occupies main memory only for components that are actually being used. Whereas the Virtual Component and Virtual Proxy patterns focus largely on creating component memory on demand, the Activator pattern focuses on a broader set of issues, such as locating services and activating/deactivating service execution contexts on demand.

The *Broker* architectural pattern [POSA1] structures distributed software systems with decoupled components that interact via local and/or remote invocations. A broker component is responsible for coordinating communication, such as establishing connections and forwarding requests, as well as for handling results and exceptions. Remote objects represent services that reside in servers. For performance and scalability reasons, these Broker systems often instantiate the Activator pattern to spawn server processes on demand. A common example is the Implementation Repository in CORBA-based ORBs [VH99].

The *Lazy Acquisition* design pattern [POSA3] defers the acquisition of resources late in the system lifecycle, e.g., at installation- or run-time. Although this pattern is similar to the Activator pattern, these patterns address different problem contexts at different levels of abstraction. The Lazy Acquisition pattern defines a broad strategy for allocating resources, such as shared, passive entities like memory or connections, to active entities, such as services. Activator, in contrast, is a more focused pattern that addresses the activation and deactivation of service execution contexts and services in resource-constrained distributed computing environments.

The small memory patterns in [SmallMemory] describe a range of other techniques that can be applied to reduce the consumption of memory in embedded systems and handheld devices with their limited computing horsepower.

References

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

[JavaRMI] W. Grosso, *Java RMI*, O'Reilly, 2001.

[PLoP9] A. Corsaro, D. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications," *Proceedings of the 9th Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, September, 2002.

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: *Pattern-Oriented Software Architecture – A System of Patterns,* John Wiley & Sons, 1996.

[POSA2] D.C. Schmidt, M. Stal, H. Rohner, F. Buschmann: *Pattern-Oriented Software Architecture, Volume 2 – Pattern for Concurrent and Networked Objects,* John Wiley & Sons, 2000.

[POSA3] M. Kircher and P. Jain: *Pattern-Oriented Software Architecture, Volume 3 - Patterns for Resouce Management,* John Wiley & Sons, 2004.

[Rago93] S. Rago: *UNIX System V Network Programming*, Addison-Wesley, 1993.

[SOAP] E. Newcomer, *Understanding Web Services, XML, WSDL, SOAP, and UDDI*, Addison-Wesley, 2002.

[Ste90] R. Stevens, *UNIX Network Programming*, Prentice Hall, 1990.

[VH99] S. Vinoski and M. Henning: *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.

[SmallMemory] C. Weir and J. Noble, *Small Memory Software*, Addison-Wesley, 2000.