Dependency Graph Pattern

Thomas Reicher, Asa MacWilliams, Bernd Bruegge
{*reicher,macwilli,bruegge*}*@in.tum.de*

Institut für Informatik, Technische Universität München
Boltzmannstraße 3, D-85748 Garching bei München, Germany
Phone: +49 (89) 289-18206, Fax: +49 (89) 289-18207

Abstract. The Dependency Graph pattern handles the recursive dependency of resources by introducing dependency descriptions, which model a resources' dependencies on other resources. Before a resource is acquired, its dependencies and recursively the dependencies of required resources are evaluated by a resource manager to set up a graph of recursively dependent resources.

1 Example

Using one resource often means using many resources, because the initial resource depends on other resources. This dependency often continues recursively (Figure 1). A resource can be anything that is required for a user service or application, for example, a runtime software or hardware component, or a shared library.

For example, in a pipes-and-filter architecture for image processing there is a processing chain with an image source, possibly filters, a feature detector, and a feature comparator. The result is given to a consumer such as a feature tracker which calculates a position from the feature positions in the image. The use of the tracker resource requires the use of a whole chain of other resources. Traditionally, it is left to the developers of the individual resources to find and use required resources, either by binding them directly through libraries or by asking a resource life cycle manager as described by Kircher and Jain [3].

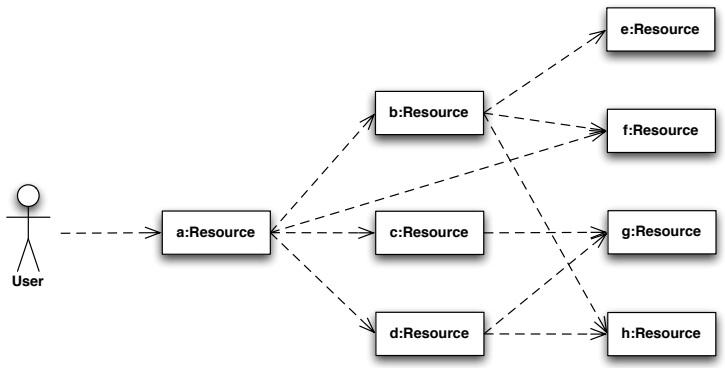

Fig. 1. A user acquires a resource which recursively uses other resources it depends on.

2 Context

The structure of a complex system, consisting of many components or resources which depend on each other, should be changeable after component development time. The time of change can be any time in the components' lifecycle after development, e.g. installation, initialization or run time. However, component developers should be shielded from the complexity this entails.

3 Problem

In order to keep the structure of the complex component-based system changeable after component development time, the principle of information hiding should be applied: resources should not know about the dependencies of other resources. This shields component developers from the complexity of the entire system, and allows components to be replaced.

However, hiding the dependencies is at conflict with keeping the system manageable. If dependencies between resources are hidden, several issues arise. Some are already covered by the Resource Lifecycle Manager pattern, but some go beyond it:

- Availability—Some resources are shared by several dependent resources. If such shared resources are called individually by different resources, it is difficult to implement a flexible resource sharing policy.
- Obscurity—Developers of a calling resource must write a reasonable error message when a resource cannot be acquired. Often, when a necessary resource somewhere in the dependency chain cannot be found, the application refuses to work without providing a proper explanation. Indeed, such a message may not be available to the application, since the error is not propagated back up the dependency chain.
- Manageability—If each collaboration between individual resources is managed by the resources themselves, there will be heterogeneous solutions preventing a general management of the resources and their collaborations.
- Dependency cycles—Cycles in a dependency graph, which could possibly lead to initialization deadlocks or endless loops, are harder to detect and to solve if such chains are set up without a coordinating instance.
- QoS enforcement—The quality of service that a resource promises might depend on the quality of service it gets itself from other resources. Therefore it might be needed to ensure QoS along the chain of interdependent resources.
- Implementation exchangeability—Often developers of a resource want to use a resource of a particular *type*, not a particular instance. Indeed, in dynamic systems such as mobile systems, the available instances of a particular resource type can change. Then there must be a managing instance that replaces unavailable resource instances by new ones.

4 Solution

To make the implicit and hidden dependencies explicit and visible, introduce externally accessible *Dependency Descriptions* for each resource. Such a description must at least store a list of other resources the resource depends on. Introduce a separate *Dependency Repository* which stores the Dependency Descriptions. Additionally, use a *Dependency Manager* that reads the Dependency Repository and calculates a dependency graph, with the resources as nodes and the possible usage relationships as edges. Based on this graph, the Resource Lifecycle Manager establishes connections between interdependent resources. In addition, it can load and start the resources on demand.

5 Structure

The participants of the Dependency Graph pattern are shown in Figures 2 and 3.

A *Resource* is an entity that provides services to a Resource User or to other Resources.

A *Resource User* acquires and uses resources as a proxy for the real end user. It is a special resource that only requires other resources but does not provide any of its own.

A *Dependency Description* lists the other resources that a resource depends on. In the simplest case, this is a list of required resources. A more complex case is described in Section 12.

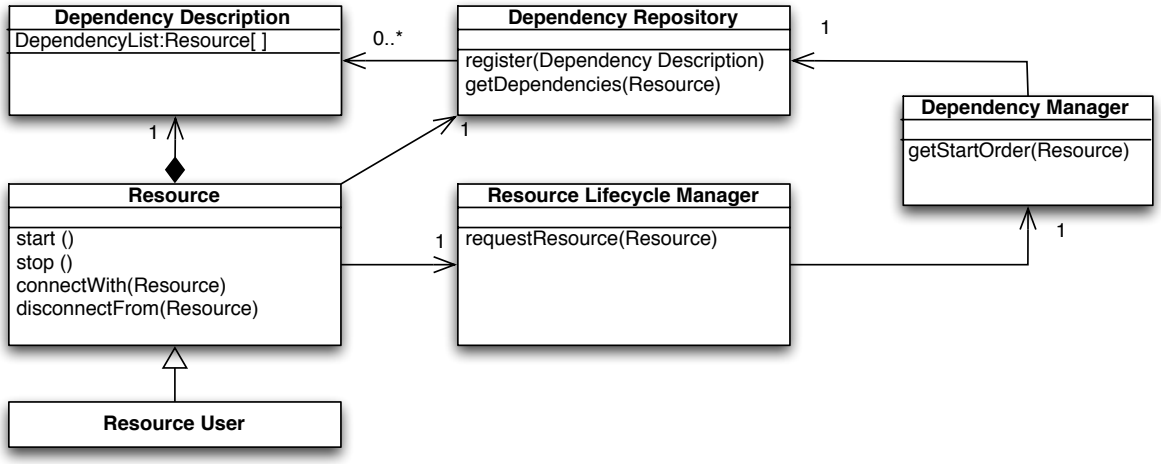

Fig. 2. Class diagram of the Dependency Graph pattern.

Class Resource User	**Collaborator**
Responsibility • Acquires and uses resources for the end user.	

Class Resource	**Collaborator**
Responsibility • Represents a reusable entity. • Is managed by the Resource Lifecycle Manager. • Registers its dependencies with the Dependency Repository.	• Dependency Repository • Resource Lifecycle Manager

Class Dependency Description	**Collaborator**
Responsibility • Describes the dependencies of a resource.	

Class Dependency Repository	**Collaborator**
Responsibility • Stores the Dependency Descriptions.	

Class Resource Lifecycle Manager	**Collaborator**
Responsibility • Calculates a dependency graph for a Resource. • Manages the Resources.	• Resource • Dependency Repository

Class Dependency Manager	**Collaborator**
Responsibility • Calculates the recursive dependencies among the resources.	• Dependency Repository • Resource Lifecycle Manager

Fig. 3. Classes of the Dependency Graph pattern.

A *Dependency Repository* stores the dependency descriptions of resources.

A *Resource Lifecycle Manager* manages the lifecycle of resources and establishes connections between resources. It requests a start order of dependent resources from the Dependency Manager for the required resource a user needs, and initializes and connects them.

A *Dependency Manager* identifies the dependencies among resources on the base of the Dependency Descriptions and calculates a start order among them. The start order is handed over to the Resource Lifecycle Manager.

6 Dynamics

The pattern consists of the following activities (Figure 4):

- Each Resource registers itself with the Dependency Repository by providing a Dependency Description. Alternatively, the Dependency Description may be provided by an installation tool, e.g. when installing a new resource.
- The Dependency Repository stores these dependencies.
- The Resource User needs a Resource and requests it from the Resource Lifecycle Manager.
- The Resource Lifecycle Manager contacts the Dependency Manager to receive a start list of all Resources the requested Resource depends on directly or indirectly.
- The Dependency Manager contacts the Dependency Repository to receive a list with all Resources the requested Resource depends on.
- For all Resources in this list, it recursively requests more Dependency Lists from the Dependency Repository.
- The Dependency Manager calculates a start order of all required Resources
- The Resource Lifecycle Manager starts all required Resources in the calculated order, and connects them.
- The Resource Lifecycle Manager connects the Resource User with the requested Resource.

7 Implementation

Several steps are necessary for the implementation of the Dependency Graph pattern.

1. Define a static Dependency Description format, e.g. an XML file, and/or a run-time interface to create and modify Dependency Descriptions. All managed resources must be described uniformly in the same format for the Dependency Manager. It must be able to compare them and find matching resources. For example, it must be possible to decide weather a resource provides a service which another resources requires by using the same name for the service. Simple resources might provide only one service with a particular standardized name; more complex resources could provide several services.
2. Create a Dependency Repository which stores the Dependency Descriptions for all managed resources. The Dependency Repository reads Dependency Descriptions and makes them accessible at runtime for the Dependency Manager. For simple cases, there is one Dependency Repository for one Dependency Manager in a system. In larger systems there might be several Repositories which can be used by several Dependency Managers in a distributed system.
3. Define resource management semantics as described in the Resource Lifecycle pattern.
4. Define resource access semantics, based on the interaction of the resources.
5. Define resource creation semantics: specify whether a resource can be started several times, only once (Singleton), whether resources can be managed in a resource pool, etc. This information is used by the Resource Lifecycle Manager.
6. Define graph topology constraints such as the interdiction of cycles.

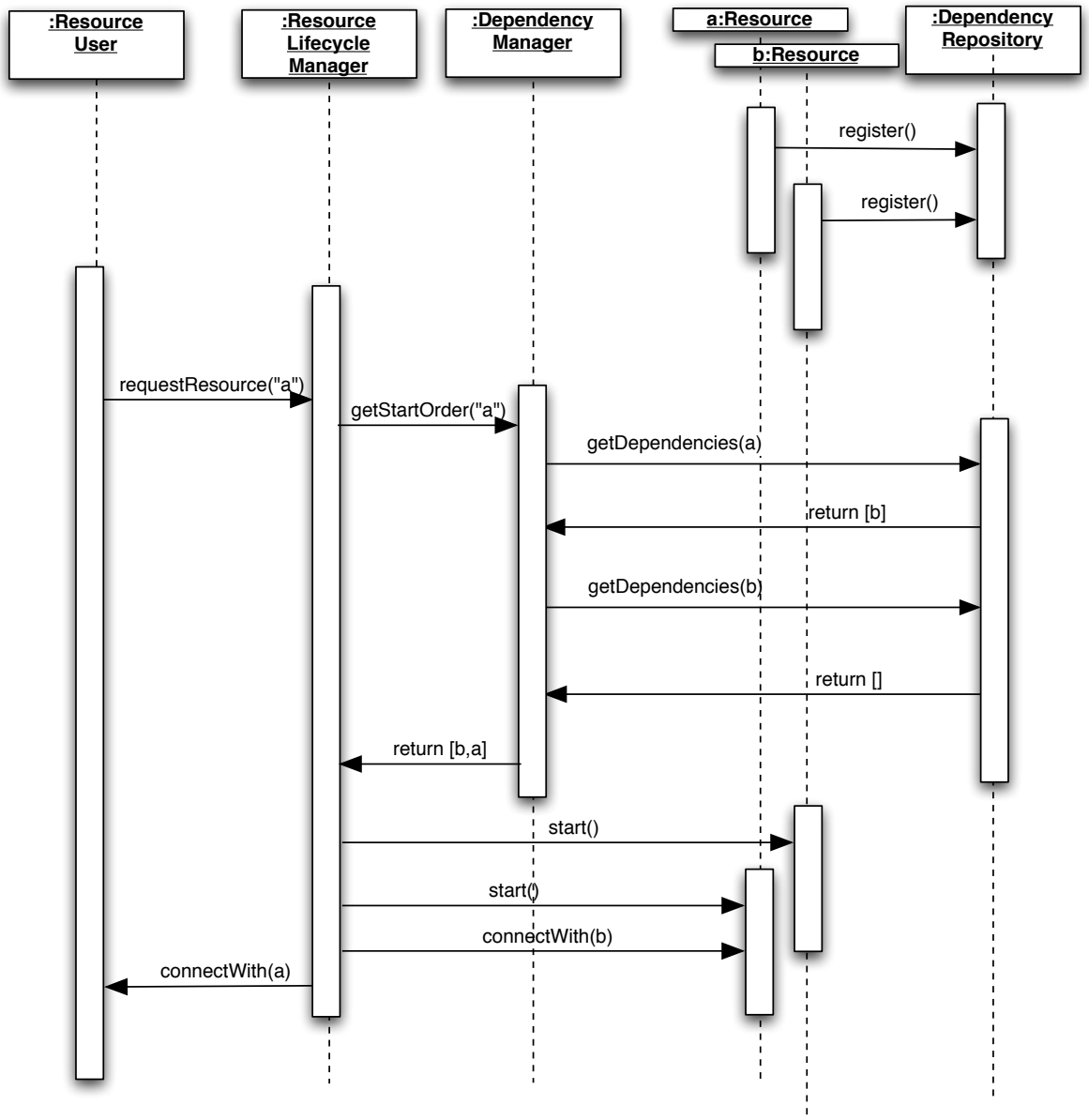

Fig. 4. Sequence diagram showing the use of Resource a, which depends on Resource b. The upper part shows the registration of Resources with the Resource Repository, the lower part the creation of the graph of dependent Resources (here a very short one with just Resources `a` and `b`).

To use this implementation within an existing system, the following steps are necessary:

1. Determine the list of required resources and their dependencies.
2. Create a Dependency Description for each resource.
3. Modify the initialization code of the resources so they can be managed by a Resource Lifecycle Manager.
4. Modify the application to acquire resources through the Resource Lifecycle Manager.

8 Consequences

Benefits of the Dependency Graph pattern are:

- Design Visibility—This pattern shows which other resources a resource depends on, not only directly but recursively. This makes deploying systems and diagnosing errors easier.
- Offline precalculation of dependency graph—The description of resource dependencies of the resource in the repository allows the dependency graph to be precalculated and resources to be started more quickly.
- Separation of concerns—The system setup (resource location and connection) is separated from the steady state of resources using each other. Thus, system setup and steady state can be optimized separately. For example, setup could be flexible but slow, whereas communication in the steady state is highly efficient.
- Resource allocation—The knowledge about the dependencies among the resources allows the implementation of different resource allocation strategies based on an overview of the resource demands.

Liabilities of this pattern include:

- Compatible resource descriptions—In order to set up a graph of interdependent resources from a pool of resources, the descriptions of them must be compatible, i.e. different developers of resources must agree on the same names for resources and attributes. This is difficult in particular for open systems with several unrelated developers.
- Adequate resource descriptions—The calculation of a dependency graph on the base of resource descriptions requires a descriptions language that is powerful enough to describe resources very precisely. The dependency manager must be able to find resources that indeed match without bad matches.
- Compatible resource implementations—Compatible resource descriptions are the base to find matching resource pairs. However, there is always the gap between resource description and resource implementation. This is mainly the case in open systems with independent resource providers. This problem is still unsolved for open systems.
- Development complexity—There are no development tools that support the developer of a resource with modelling its Dependency Descriptions. Usually, such a description is not part of the development environment for the resource functionality but must be done with additional tools (in the simplest case a text editor).
- System complexity—A Dependency Manager adds complexity to a system. A dependency manager is a powerful component that can make the developers life easier because he does not have to care about recursive dependencies. On the other hand, as a central component it must be implemented very carefully because all resources depend on it. This pattern is most useful in systems with several nested interdependent resources such as operating systems, distributed systems or highly dynamic systems. On the other hand, if badly developed it might be a single point of failure.

9 Limitations

Several aspects are explicitly beyond the scope of this pattern.

- This pattern does not address stateful resources. The dependencies between resources are modeled without regard to resource state. To address problems such as dynamic failover between components at run time, the modeling of component state would become necessary. This would require a complex component description language.
- Resource multiplicity and identity are not handled by this pattern. If one resource depends on several instances of another resource, the dependency descriptions must be extended to handle this, and to allow different instances of the same resource.

10 Known Uses

OSGi framework—The OSGi framework[8] is a specification for the delivery of multiple services over wide-area networks to local networks and devices. Part of the framework is a Resource Manager which recursively starts all required supporting services for a requested service. Therefore each OSGi service provides a description of other services that must be started in advance.

Linux RPM—The Linux RPM [1] file format provides a list of the other packages a package depends on. The Package Manager reads that list, checks if all required packages are already installed, and issues an error message if some of them are missing. More advanced in handling packet dependencies is the Debian Package Manager, which can install recursively required software packets automatically.

DWARF—The middleware for the DWARF augmented reality framework [2] uses XML-based descriptions of DWARF services to setup a chain of interdependent services [4]. Here, the dependency repository and resource lifecycle manager are implemented as one distributed component, the *service manager*.

Windows Service Management—Windows Services must described in a registry entry. Among other information, they describe their name, the name of required services and name of the executable to start them. When a service should be started, the Windows Service Management calculates the graph of the required other services that service depends on. Based on this list, it starts any services that currently are not running.

Linux kernel modules—Linux maintains a separate modules.dep file that specifies module dependencies. This list is used to load required kernel modules on demand.

11 Variants

There are three implementation variants of the Dependency Graph pattern. The first two are invisible to the resource user and the resources themselves, but are different in the implementation.

Combine Dependency Manager and Resource Lifecycle Manager components The Dependency Manager and the Resource Lifecycle Manager can be combined into one component that is responsible for both tasks. There is no difference for the resource user to the original solution since the Resource Manager is the facade for it. However it is different from the implementors point of view. It makes the implementation easier because there is one component less. However, it is also less flexible as it does not allow to use exchange the Dependency Manager implementations supporting different management strategies for resolving conflicts or breaking cycles.

Distributed repository and dependency management The Dependency Repository and the Dependency Manager can be broken into several federated components in a distributed system. The distributed approach makes the system more flexible and avoids a single point of failure. It also allows the integration of service location mechanisms, integrating newly discovered resources into existing dependency graphs at run time. This variant is used in DWARF[2]. However, this solution is more complex as it requires the development of a protocol between the participating manager components.

No Resource Lifecycle Manager The main use of the pattern is to provide a resource to some other resource and to check and resolve any internal dependencies beforehand. The instantiation of these resources can as well be left over to the resource requestor. Then it only gets a dependency list from the Dependency Manager and is responsible for acting accordingly. This solution can be used if a common resource management is not possible or needed, for example if the goal is only to check if a resource can be used and which resources needed to be installed.

12 Extensions

The Dependency Graph pattern can be extended in several directions, taking advantage of the existence of explicit Dependency Descriptions.

Abstraction: services and quality of service. In the simplest case, a Dependency Description simply lists the other resources a resource depends on. This means that the dependencies refer to specific software or hardware components. For greater flexibility, it is possible to specify the abstract service types a resource provides. For example, the Dependency Description of an application that displays its on-line documentation in HTML format could refer to the abstract service type `WebBrowser` rather than the concrete resource `Netscape`.

In this extension, a Dependency Description becomes more general. It does not only include a resource's required services, but also the services it provides to other resources.

In order to distinguish between different implementations of a service, the resource's provided services may be described using QoS attributes. In the same fashion, the required resources can be specified more closely using boolean predicates over these attributes, or numeric expressions for optimization.

This extension is used in DWARF, where components describe what they provide by *Abilities* and what they require by *Needs*. Abilities have attributes which can be matched by the needs' boolean expressions. It is also used in RPM, which has *requires* and *provides* fields in package descriptions. Simple attributes and predicates are available in RPM, e.g. to specify a minimum compiler or API version.

The introduction of service types can be used in open and mobile systems with service discovery to adapt the system to changes in the availability of services over time. But such a solution is more complex and less robust. The overall problems are service naming and compatibility. First, service user and service provider must agree on the same name for a particular service. This could be solved by service specifications by organizations such has the IETF. The second problem is that once a service provider has been found, it is not clear if it really implements what it promises. This problem is typical for open systems.

Support for Design by Contract. In Design by Contract [5], preconditions for the use of software resources are made explicit, as well as postconditions that hold after use, if the preconditions have been met. These preconditions and postconditions are specified explicitly, usually taking advantage of special programming language features. The preconditions and postcondition create a *contract* between the caller of a method and the object implementing that method.

A resource's dependencies on other resources can be modelled as a simple contract. The method in question is a resource's `start()` method. Its precondition is the availability of other resources, as specified in the Dependency Description. its postcondition is the availability of the resource itself.

Thus, a contract exists between the Resource Lifecycle Manager and each resource. Only if the Resource Lifecycle Manager can provide all required resources with the required quality, then a resource can start and provide its own services.

Beyond that, the resource lifecycle manager sets up a graph of simple contracts between the resources themselves when it initializes and connects them. These contracts are based on the same simple preconditions, the availability of a resource and its services.

Using attributes and predicates to model quality of service as described above, the preconditions and postconditions can become more complex. Thus, an extended version of the Dependency Graph pattern can benefit from a design by contract approach, where the preconditions and postconditions are made explicit in the form of Dependency Descriptions. A general solution to negotiating collaborations between distributed designed-by-contracts software components is an area for future research.

Support of component-specific communication methods. Traditional middleware technologies such as CORBA [7] or COM+ [6] do not support heterogenous means of communication, e.g. video streams and method calls. To adapt the system's transport facilities to the requirements of the resources, the Dependency Descriptions can include the supported communication mechanisms. The Resource Lifecycle Manager can then establish appropriate communication channels. This approach is used in DWARF.

13 See Also

Design by contract was first introduced by Bertrand Meyer [5]. He uses it to specify assertions on objects in Eiffel.

Resource Lifecycle Manager describes a pattern that decouples the management of the lifecycle of resource from their use [3]. The Dependency Graph pattern extends this pattern by targeting recursively dependent resources.

References

[1] E. Bailey, *Maximum RPM*, Sams, 2003.
[2] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner, *Design of a ComponentBased Augmented Reality Framework*, in Proceedings of ISAR 2001, 2001.
[3] M. Kircher and P. Jain, *Resource Lifecycle Manager*, in European Pattern Language of Programs conference, Kloster Irsee, Germany, 2003.
[4] A. MacWilliams, T. Reicher, and B. Brügge, *Decentralized Coordination of Distributed Interdependent Services*, in IEEE Distributed Systems Online – Middleware Work in Progress Papers, Rio de Janeiro, Brazil, June 2003.
[5] B. Meyer, *Applying Design by Contract*, IEEE Computer, 25 (1992), pp. 40–51.
[6] Microsoft Corporation, *COM+ Component Model*. `http://www.microsoft.com/com`, 2003.
[7] Object Management Group, *CORBA 2.4.2 Specification*. formal/01-02-33, 2001.
[8] OSGi Alliance, *OSGi Service Platform, Release 3*, Mar 2003.