Building EJB Applications - A Collection of Patterns

Eberhard Wolff, Alexander Schmid, Markus Völter
MATHEMA AG, Germany
{eberhard.wolff|alexander.schmid|markus.voelter}@mathema.de

Version 1.4, 2001-07-18

Abstract

Although Enterprise JavaBeans provide simple APIs for relatively complex tasks, designing and implementing a scalable, maintainable and reasonably fast application based on EJB is not trivial. Over time, a set of proven patterns has emerged – this paper presents some of them. The patterns will be part of a book about EJB architecture and application patterns which is currently being written by the authors of this paper. We do not address other parts of J2EE such as Servlets or JMS, we solely focus on the "middle tier" made up of EJBs.

Intended Audience

These patterns are intended for architects, designers and implementers of EJB applications. We expect the reader to understand the EJB architecture and APIs, and we expect that the reader has some experience in building EJB applications.

Pattern Form

This is a collection of patterns for the area of Enterprise JavaBeans. They do not form a pattern language which explains how to build components from scratch. They provide pin-point help for specific aspects of component development with EJB.

The patterns use a slightly modified Alexandrian pattern form. For brevity, a description of this form is omitted in the submission.

We have used three different forms for the resulting context section of the patterns. We'd appreciate feedback, on which works best.

Cited Patterns and Principles

This paper is a part of a book we are currently working on. The book contains an explanation of component-based systems in general and introduces some patterns and principles to do so. As these patterns are not present in this paper, we present a short overview of the patterns cited herein. The full patterns can be downloaded from www.voelter.de/cpl. They have been workshopped at EuroPLoP 2001, which means they can also be found in the proceesings.:

Separation of Concerns

This is a principle with the goal of separating functional parts of a software architecture from technical. This way the developer of one part of the system need not care about the other part of the system. In EJB this shows for example in the different roles (Deployer, Component Developer, Container Provider etc.) with their different responsibilities.

Container

To reach a Separation of Concerns most of the technical aspects of the components are implemented in the Container. The term Container is also used in EJB to denote the software that the components run in. It offers certain services for the components and also takes care of certain aspects of the component i.e. it calls the lifecycle methods.

Component

To enable reuse and to make parts of the system independent from other parts Components are introduced. They partition the functionality of the system, depends only on other Components' interfaces and is reusable in isolation. In EJB three types of Components exist: Entity Beans, Stateless and Stateful Session Beans

Component Interface

To decouple Components from each other and to make implementations interchangeable Component are only accessed by their interface. In EJB this is the Remote Interfaces that defines all methods that might be accessed by a client application or another component.

Component Implementation

To define how the Component Interface should be implemented the Component Implementation is used. In EJB this is the implementation class i.e. the one that extends SessionBean or EntityBean repectively.

Primary Key

Each persistent component must have a unique identifier to allow later retrieval of the same component. This unique identifier is called Primary Key. In EJB this is directly implemented as the Primary Key of an Entity Bean.

Restricted Implementation

Because the component should only implement the functional part of the softeware the implementation of the component is usually restricted. So the component can not interfere with the technical concerns the Container takes care of. In EJB the component must not do any thread manipulation, file access or server socket connections for example.

Interception

When a component is called the Container is usually responsible to enforce certain policies concerning security or transactions for example. Thus each call must be intercepted to accomplish this and is forwarded afterwards to the Component Implementation. In EJB the security and transaction features are implemented in this way.

Configuration Parameters

To adapt a component to different usage scenarios parameters might be set to certain values. These parameters must not be defined in the code because the component should be adapted without changing code. In EJB Configuration Parameters are defined in the Deployment Descriptor.

Annotations

Access control lists and transaction attributes for methods are a technical concern and should not be included in the code. Also they tend to be changed quite frequently. So they must be defined as Annotations in a separate place. In EJB the Annotations are part of the Deployment Descriptor.

Architectural Patterns

Quick Access Table

if..	use..
... Entity Beans are too heavy-weight for you entities i.e. you do not need synchronization	TYPE MANAGER: A Stateless Session Bean directly accesses the databases
... you must implement a Singleton as a unique point in your system	DISTRIBUTED SINGLETON: An Entity Bean or a component with synchronization through a database or an External Service
... a business process needs collaboration and/or longevity and thus can not be implemented as a Session Bean	PROCESS AS ENTITY BEAN: Instead of a Session Bean use an Entity Bean that offers the required features
... a service can not be implemented because of EJB restrictions e.g. no access to threads or files	EXTERNAL SERVICE: Use an external application for the restricted parts.
... collaborations between or transaction across Entity Beans must be defined	SESSION BEAN FACADE: An additional Session Bean calls the Entity Beans and thus defines collaborations and transactions
... long transactions occur in your application	LONG TRANSACTION: A Stateful Session Bean collects the data and does the work in one short database transaction.

External Service

As you implement your Enterprise JavaBeans application you recognize the need for a service which cannot be implemented within the EJB programming model, because of RESTRICTED IMPLEMENTATION. The container contract[1] does not allow to implement the service you need.

Imagine an application that needs some kind of time based notification mechanism like a cronjob[2]. The usual way to implement this in Java is to start a thread that does the handling of the timetable and then notifies its client at the point needed. This implementation is illegal in Enterprise JavaBeans as the container contract prohibits to start your own thread in the container.

Another example is a logging component that should output log messages to a file. This can not be implemented using EJBs because direct file access is forbidden. Several other limitations exist also. For more details see RESTRICTED IMPLEMENTATION. So the service can not be implement within the Container.

Therefore:

Implement a standalone application offering the service you need. This is called a EXTERNAL SERVICE. The EXTERNAL SERVICE can be active and then calls your Enterprise JavaBeans application (e.g time based notification). In other cases it is passive and gets called by your Enterprise JavaBeans application (e.g.

1 The container contract is the sum of all the services that the container and the component offer for each other. This includes limitations of the features that might be used in the code of the component.

2 cron is a program usually used in UNIX systems to automatically start reoccuring tasks.

logging component).

***

Advantages

As your External Service is not restricted through the container contract you are free to use every feature you need within it. It is even possible to implement the external service not in Java but in a different language than Java using for example CORBA.

Drawbacks

An External Service is no EJB. So the benefits of EJBs such as high-availability and load balancing are lost. In most cases these features must be manually implemented in the External Services to avoid a single point in the systems that is neither scalable nor fail-safe. Otherwise the performance and availability of the whole system might be reduced.

Also you have to take care that it is integrated in the security and scaleability concept. This is relatively complex as the propagation of transaction and security contexts to the External Service is difficult.

Additional administration efforts are needed because not only your application servers but your services have to be deployed and administrated. Especially as your Containers administration and monitoring tools will most probably not work with the External Service this can be difficult and costly.

Variants

Some Container offer support to run an External Service within the virtual machine of the Container. This improves the performance of calls to the External Service as no more inter process calls are necessary when the container calls the External Service. As this feature is not standardized right now, of course you loose portability.

Long Transaction

In your application, some transactions last several minutes or even longer. The database system lacks support for these long running transactions. Lots of database entries are usually locked due to the long transactions. This causes long waiting times for the users and thus impacts the usability of the application.

Often a lot of work on the application level is just one transaction on database level. This is especially true for interactive applications. However, most databases are designed for short transactions only. Also a big number of database rows that are locked for a long time might make the database unusable for other clients because the locked data is not accessible to them.

Imagine a shopping cart in an ECommerce Website. During the shopping process several items are ordered. So the number of items in store must be changed in the inventory for each item. If at the start of the shopping process a transaction was begun, this means that each item is locked in the database. Then nobody else can order these item concurrently while the transaction is active. Of course this is not acceptable.

Therefore:

Provide a Stateful Session Bean that collects the data without opening a transaction on the database. So the data can be entered by the user over along period of time i.e. a shopping session on a Web site. The database work is done in one single operation when all data is collected. So for the client this resembles a long running transaction. However, the real transaction does not start until the final operation is completed. Then the collected data is written in one short transaction into the database.

***

Variants

Even longer transaction can be realized using Entity Beans instead of Stateful Session beans (see PROCESS AS ENTITY BEAN). For performance optimization it might make sense to collect the data on the client and use a Stateless Session Bean. Note that the handling of the transaction logic is distributed in this case: The collecting is done on the client while the real transaction is done on the server.

Drawbacks

To make sure that the written data is really consistent probably the database should be checked during the final operation. If someone else changed the data while the Session Bean was active the collected data might be invalid. E.g. someone else changed a database entry and the collect data still has the old entry. If this was written to the database the modification by the other transaction would be lost. In this case the an error should be sent to the client and the data should not be written into the database.

Another problem is that the isolation of the transactions might be broken if you use this pattern. Imagine that some data must be read from the database before the update part of the transaction can occur. The update is done in the final method call. The previously read values might have changed by that time because the data was not locked in the database. Thus it might be necessary to check whether the read data was changed before the transaction is completed and indicate a failure otherwise. Note that if no data is read or this kind of conflicts can be ruled out for other reasons this checking can be omitted.

If this checking is included the Pattern emulates optimistic locking. This is a technique that can be used by databases to enable long running transactions. Usually databases use pessimistic locking and lock the database rows at the beginning of a transaction for the complete duration of the transaction. In this case the transaction can be sure to succeed and write its result to the database. In the case of optimistic locking the data is not locked. Instead conflicts with other transaction are detected when the transaction is committed. I.e. a rollback occurs if other transactions modified the data that is to be written by the current transaction.

Another potential problem is that the pattern depends on a time stamp in the database table. If this time stamp is not present it is often not possible to simple add it to the existing database because of the size of the table. In this case the time stamp can be stored in an additional table that contains only the primary key and the time stamp.

Use this pattern only if your are either sure that conflicts rarely occur or the work of a transaction can be easily redone because each conflict will make the transaction fail. If you can deal with a database that is not entirely consistent you can omit the checking. In this case each transaction succeeds but some changes might be lost. This lowers the number of failures but more frequent failures are the biggest drawback for optimistic transactions compared to pessimistic transactions.

Process as Entity Bean

You have complex business processes that have a complex state, run very long and/or require the collaboration of multiple users. Stateful Session Beans offer neither the needed safety for the data nor concurrent access from more than one client. This means that your business processes can not be modeled with Session Beans.

Stateful Session Beans can be accidentally destroyed by container crashes or system exception. If a Stateful Session Beans is not reference it will be destroyed after a timeout. This means that the data stored in Stateful Session Beans is not safe. Business Processes using Session Beans contain just a few operations and it is assumed that they are called by one client. So concurrent access is not possible and it is not straight-forward to pass a reference to a Session Bean from one client to another. Often access from a single client is not enough as collaborative work flow systems and other tools show.

Therefore:

Model long running or multi-user business processes as Entity Beans instead of Session Beans. This allows your processes to be persistent and accessed concurrently by many clients.

❊❊❊

General

Even though Entity Beans were originally invented to model Business Entities some features of Entity Beans are very useful for Business Processes. For example concurrent access to Entity Beans is possible and this can be very effective for the support of collaborative work. Also the additional support for a persistent storage of the state of an Entity can be a good help to enable long-running Business Processes. Note that there are some things that show that Entity Beans are not really meant for Business Processes: An Entity Bean has an identity and is permanently stored in a database. A process needs the identity only to pass it to other clients that take also part in the process. The database is only used to make sure that the state of the process is not lost. After the process is completed this state can usually be deleted.

Usually Entity Beans use Bean Managed Persistence because the data they represent was stored in the database before the Entity Bean were developed and they need the additional flexibility to map the data in the database to the Entity Beans. As business processes are rarely represented in a database it makes sense to use Container Managed Persistence for the implementation of this pattern: The process just needs to be persistent but you need no explicit control of the database layout.

Related Pattern

This pattern can be used to enhance LONG TRANSACTION to deal with even longer transactions.

Session Bean Façade

Usually the access to Business Entities requires that certain methods must be called in some defined order or within the same transaction. Also methods of some Entities Beans might only make sense in conjunction with other Entity Beans. However, you cannot define that a transaction should span multiple methods of an Entity Bean or different Entity Beans.

The transaction attributes can only be set for each method. This means that there is no way to enforce that multiple operations or operations of multiple Entity Bean must be called within the same transaction. This is left to the client. The same is true for defined processes that span multiple Entity Beans: Such things can not be defined in a normal Entity Bean.

Therefore:

Provide a SESSION BEAN FAÇADE i.e. a Stateless Session Bean that accesses Entity Beans. As both Components are located on the server only server internal communication takes place and no network overhead will occur. In addition, one operation in the Stateless Session Bean might call several methods on different Entity Beans. A transaction can then span all methods called on the Entity Beans. Avoid direct access of Entity Beans whenever possible.

❊❊❊

General

Stateless Session Beans were invented with short lived Business Processes in mind while Entity Beans were meant to implement Business Objects. As Business Processes work with Business Objects it is natural to use Stateless Session Beans to access Entity Beans. However, this Pattern goes even further by making indirect access of Entity Beans through Stateless Session Beans the default. This is because usually the client is not interested in the Entity Beans but rather the complete Business Process as implemented by the Stateless Session Beans. As such a process most often executes as exactly one transaction the problem concerning the transactions is also solved.

Another benefit of using this pattern is that less remote calls are needed. If all the Entity Beans were directly accessed from the client each method call would be a remote call. After applying the pattern at least some of these remote calls are replaced by local calls on the server i.e. calls from the Session Bean Facade to the Entity Beans.

Related Patterns

Compared to LONG TRANSACTION this pattern implies that all parameters that are used during the transaction are known before hand while with LONG TRANSACTION they are collected and afterwards the transaction is executed. VALUE OBJECTS can be used as parameters for the FACADE and can be forwarded to the Entity Beans without further modification. This pattern appears to be the adaptation of the FAÇADE pattern in [GHJV94] to Enterprise Java Beans but has a different goal: While the original FAÇADE pattern tries to hide the complexity of a subsystem behind one object this pattern is just intended to provide a means to express the collaboration between Entity Beans. PROCESS AS ENTITY BEAN is also a pattern that tries to embrace multiple method call but with the emphasis is on collaboration to fulfill the task. LONG TRANSACTION puts the emphasis on the longevity of the transaction.

Distributed Singleton

Your Enterprise JavaBeans application needs to implement the [GHJV94] SINGLETON pattern. Through RESTRICTED IMPLEMENTATION you are not allowed to implement the SINGLETON pattern using static members and the synchronized statement. You need to find a way to implement a SINGLETON conforming with the Enterprise JavaBeans specification.

While implementing a complex application you often need to implement a service that will be accessed in a controlled way from different parts of your application. For example when you want to implement something like an id generator you need a single instance providing the ids. The SINGLETON pattern describes a way to implement a service that fulfills these needs. Unfortunately it can't be implemented easily in an Enterprise JavaBeans application. This is because through RESTRICTED IMPLEMENTATION some features of Java are not allowed to be used.

For example to implement load balancing, more than one CONTAINER might host your beans. These CONTAINERS run on different computers on different virtual machines, so a static member will not be shared by them. So every CONTAINER would have its own static id member what makes generating a unique identifier impossible.

Therefore:

Find a point in your architecture that runs with only one instance or at least behaves like this. Move your SINGLETON synchronization mechanism to this point.

❊❊❊

General

As you need a point in your architecture that behaves like it would be a single point you have different choices where your synchronization might occur:

You can use the database to synchronize. There are different ways to do this:

- Use an Entity Bean to synchronize your method calls.
  As the Container is responsible for synchronizing access to Entity Beans it behaves like a synchronization point in your application.
- Use a Stateless Session Bean that accesses the database and synchronize through the locking mechanism of the database. Also a stored procedure can be used here. This way also external systems could be synchronized with the EJB application. This is a way to synchronize for example with legacy systems. Stored Procedures can be used to implement this, too.

Alternatively, you can use an EXTERNAL SERVICE holding a "usual" SINGLETON implementation. Within the EXTERNAL SERVICE the "usual" SINGLETON implementation is valid, because RESTRICTED IMPLEMENTATION does not cover the EXTERNAL SERVICE.

As the different approaches have different features the concrete implementation and architecture of your system needs to be checked to see which way to implement the SINGLETON fits your needs most.

Drawbacks

The problem with SINGLETONS is of course that you lock all parts of your system that try to access your SINGLETON concurrently. This might hurt the scaleability of your application. So try to avoid SINGLETONS as often as possible (in our id generator example from above for example you might use number circles).

Also as you have to find/introduce a unique point in your application to synchronize your callers, take care not to introduce a single point of failure! Otherwise the high availability of your system will not be guaranteed anymore.

Type Manager

Entity Beans provide concurrency synchronization, pooling, and an extensive lifecycle management. These features impose some performance penalty. For applications, where many clients access a set of entities concurrently and these additional features are used, the performance penalty is acceptable. However, in systems where these features are not required, Entity Beans don't deliver optimum performance. This is particularly true for highly concurrent read access to your data.

You implemented a typical stock trading system. Many stock symbols are managed by the system. Most of the operations implement some kind of monitoring on the stocks and are therefore read-only. No locking or other concurrency synchronization is required here, because in read-only situations, no conflicts will occur. So, using Entity Beans will imply a lot of overhead which is not required.

Therefore:

Use a SESSION BEAN to work directly on the entities in the database. Use VALUE OBJECTS to represent the data in each call, together with the PRIMARY KEY to identify the entity to which the operation should apply.

❊❊❊

Advantages

You get higher performance than accessing Entity Beans because the CONTAINER does not need to instantiate and support the single Entity Beans. Your systems performance is directly linked to the performance of your database system.

Drawbacks

As you don't use Entity Beans as promoted by the Enterprise JavaBeans specification you implement an architecture that does not benefit from some features a CONTAINER might offer. For example Entity Beans using Container Managed Persistence might be cached by the container and therefore have performance advantages in some cases over a TYPE MANAGER or a Bean Managed Persistence implementation for persistence.

Variants

A TYPE MANAGER based architecture is also useful if your COMPONENTS are used to provide an interface to legacy applications which provide a procedure-oriented API. Even if the legacy application has a different interface - a Message-Oriented-Middleware for example – it might make sense to give a TYPE MANAGER as interface so that access to all systems is uniform.

TYPE MANAGER is a replacement for Entity Beans through Session Beans. So you have to deal with entity handling using the LIFECYCLE of a Session Bean.

Related Patterns

If your performance problems do not result from the concurrent usage of your entities or the distribution of your workset but from write operations on large numbers of ENTITY BEANS you should use ENTITY BULK MODIFICATION instead.

Dependency Management Patterns

Quick Access Table

if...	use...
...Entity Beans can not be reused because they have references to each other	PRIMARY KEY AS REFERENCE: Instead of direct references store the primary key
...Relations between Entity Beans are complex and change frequently	RELATIONSHIP SERVICE: Store the references in an external service instead of the Entity Bean itself
...circular dependencies make reuse and maintenance hard	EVENT LISTENER: in one direction method calls are used in the other events are sent.

Primary Key as Reference

As your Entity Beans have direct references to each other, you can not really use one of them in isolation. At compile-time you must have at least access to the COMPONENT INTERFACES of the other Entity Beans as well. It is also not possible to substitute one Entity Bean with an equivalent Bean with a different interface.

Entity Beans are COMPONENTS. This means they should not depend on other COMPONENTS. As soon as you use the COMPONENT INTERFACE of another COMPONENT these COMPONENTS depend on each other technically i.e. you must have at least the Interface to actually use or compile the referring COMPONENT. This is already sufficient to make a COMPONENT unusable on its own or in a different context where the other COMPONENTS might not be accessible. This is a major limitation on the reuse which is the major goal for using COMPONENTS. Using a Handle does not solve this problem: You must still have access to the interface of the referenced bean and the Handle is application server specific which can be another drawback.

Therefore:

Use the PRIMARY KEY instead of the reference or Handle to model the relation. As every Entity Bean must have a PRIMARY KEY, this is possible with every type of Entity Bean. This make the Entity Beans loosely coupled: They still depend on other components logically but not in term of software artifacts.

❊❊❊

General

Lose coupling is probably the most important goal for COMPONENTS. Of course COMPONENTS must be able to reference each other. However, it should be done in a way that avoid unnecessary technical dependencies.

The idea of storing primary keys instead of object references to accomplish this is a violation of object-orientated principles because an object-oriented design tries to build model consisting of objects and references between them. Nevertheless it is a good compromise for component-based systems. An interesting point to note is that the persistence for Entity Beans must use either primary keys or Handles to store references to Entity Beans. So on the database layer only primary keys and handles exist anyway.

Drawbacks

Note that Entity Beans loose the ability to directly access the Entity Bean that they reference. However, operations that work on multiple Entity Beans should be implemented using Session Beans. These are responsible for retrieving the referenced Entity Beans using the primary keys.

This leads to the problem that an Entity Beans itself can not retrieve these references. So for example it is not possible for an Entity Bean to delete other dependent Entity Beans. However, dependent objects of this type should really be implemented as DEPENDENT OBJECTS Note that Entity Beans should be coarse grained and thus be self-contained. The need to access related objects should not show up. However, using this pattern denies Entity Beans to call any method on other Entity Beans it has references to. This is also the reason why the pattern can hardly be used in Session Beans: Session Beans must actually use the referenced Beans and this is means they depend on the referenced Beans anyway.

Related Patterns

If you need to add references at run time you can create a Session Bean that manages all references in the system (a RELATIONSHIP SERVICE). This would have operations to add and retrieve relations between Entity Beans.

Relationship Service

Whenever an entity has a relationship to another entity, it must store its PRIMARY KEY (or keys, for 1:n relationships) and must provide operations to access the related entities. In applications, where you have many different relationships and these relationships change very often, you have to change the entity implementation whenever you need to add a new kind of relationship. This requires modifications in the COMPONENT INTERFACE and the implementation, recompilation, an adaptation of the table structure, and redeployment.

For example, take a product management application. Product items may be packaged in different entities, such as palettes, packets with 10, 20, 50 each, etc. Imagine you store a reference to a specific palette in a product entity, to track which delivery it was part of. If now the produces changes its packaging to containers, or trucks, or whatever, you have to adapt your entities to be able to reflect these changes: References to containers, or trucks have to be added to the entity. New dependencies are created, and operations to modify the relationship must be added to the entity.

It becomes even more difficult, when a product can be delivered either on a palette, or in a 20-pack or on a truck (or maybe even the 20-pack is on palettes on a truck...). If you would have to change your entities whenever this kind of relationship changes, you would frequently have to redeploy your application, creating a huge overhead.

Therefore:

In systems where the types of relationships change frequently, externalize the relationships into a Relationship Service. Such a service can store relationships among any two (or more) entities. It stores the entities, a type of the relationship, and even additional attributes belonging to the relationship, and not to any of the entities.

***

Advantages: Because the relationships among entities are not part of the entities themselves anymore, changing the relationship structure does not affect the entities at all. (By the way: If you look at the problem from a conceptual view, it is questionable whether it makes sense at all to be able to ask a product item on which palette/truck/etc. it was delivered. So externalizing these relationships is even more natural than keeping them in the entity.)

Using a relationship service reduces the entities to be real entities, and not places to store relationships among entities which don't belong their naturally. Because relationships are often use-case specific, this pattern enhances the chances for reuse significantly.

Drawbacks and Caveats: Entities are not completely self-contained anymore. They can only be used successfully, if a Relationship Service is available. The programmer has to access this Relationship Service explicitly, which complicates the programming model somewhat.

Variants: A relationship service can have additional benefits: You can store a list of name-value pairs with the relationship, you can give each relationship a type, and you can even predefine which relationship types are allowed for which kind of entities.

The relationships are maintained by an generic external component, using separate tables in the database. As a matter of convenience for the programmer, COMPONENTS can provide operations like *getRelationships(type)*, which uses the Relationship Service component internally, which hides the presence of the extra component.

A Relationship Service can be implemented as a COMPONENT running in the same address space as the client using it. If this is not possible because of restrictions of the Component Model, an EXTERNAL SERVICE can be used.

Related Patterns: Another way to reduce the formal dependencies on the actual COMPONENT INTERFACES of related COMPONENTS is to use PRIMARY KEY AS REFERENCE.

Please note, that the OMG already defines a relationship service for CORBA objects [OMGREL]. It can serve as a good role model, although it has many features that are not essential in many systems.

Event Listener

You don't want to have is mutual, or circular, dependencies because this has very bad consequences regarding maintenance and deployment, and it reduces the reusability of the COMPONENTS. Dependencies should always be unidirectional only – resulting in a "layered system" [POSA]. In layered systems dependencies exist only in one direction, namely from higher to lower layer. Lower layers are never allowed to directly access higher layer. But then, how do you communicate information from lower layers to higher layers?

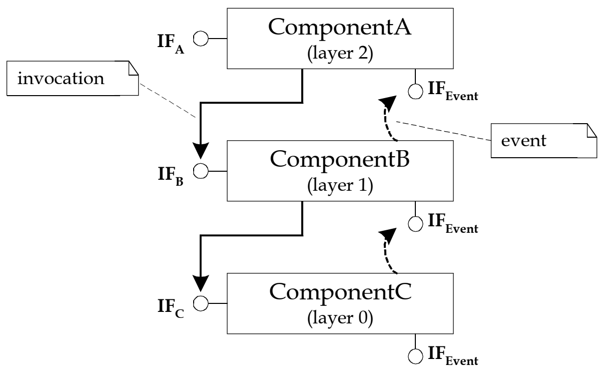
Consider a *Customer* bean and a 1:n related set of *Order* entities. The *Customer* has a flag that determines its reliability, depending on whether he paid promptly for all issued orders. Whenever an *Order* changes its payment state to *overdue*, the associated *Customer* should be notified, to adjust its reliability flag.

An order is an entity you will want to use in different systems, i.e. it should be reusable. A Customer will usually depend directly on the *Order* component, because the Customer should provide operations like *listOrders()* or *listOpenOrders()*.

If in addition the *Order* would directly depend on the *Customer* interface, you could only use the *Order* with this *Customer* component and vice versa. Reuse is compromised. It gets even worse, if you notice later that other Components in your system are also interested in being informed if an order is set to overdue – maybe the financial department wants to send out a reminder to the *Customer*.

Therefore:

Ensure that direct dependencies exist only in one direction. In this direction, use direct method invocations. For the way back, use event-like communication based on a generic event-receiver interface. Receivers of events register with the producer. Receivers are notified if an event is published.



❊❊❊

Advantages: The event publisher relies only on a generic event listener interface, not on the concrete COMPONENT INTERFACES of the receivers. This reduces dependencies to be unidirectional. For more information on this technique see [BCF].

Drawbacks and Caveats: Note that excessive and uncontrolled use of this pattern can cause major performance degradation: the invocation of an operation on one COMPONENT can result in a cascade of events being fired – without the invoker of the original method seeing it directly. Of course this is also true if you don't use this pattern and invoke business methods directly in the implementation of another method.

Also note, that this pattern requires that the producer of an event keeps a list of event receivers, possibly grouped by the event type. The COMPONENT must provide attributes and operations to manage these relationships. Because these relationships are the same for any COMPONENT publishing events, a RELATIONSHIP SERVICE can be used to keep and manage these relationships.

Variants: The events can be delivered synchronously or asynchronously. For small systems, synchronous callback is feasible. For larger systems with many dependencies and thus potential events, an external, asynchronous event distributor (usually an EXTERNAL SERVICE) should be used, as described in ASYNCHRONOUS EVENTS. In EJB, you have to be especially careful, since direct callbacks to the invoker of an operations are not allowed by default. To overcome this problem, in some CONTAINERS COMPONENTS can be marked as *reentrant*, thus allowing callbacks to the invoker of the currently active method. If this is not possible, use an external service to deliver the evens to the caller.

Additional Information: It is important to note that this pattern is not in any way specific to Entiy Beans. Session Beans can also act as consumers and producers of events.

Note that Message Driven Beans and JMS from EJB 2.0 do not directly help here. This is because Message Driven Beans are *only* message driven. As a consequence, it is not possible that a bean has a normal business interface *and* can consume events from a queue or topic. Of course it is possible to create adapter-MDBs that consume events and forward the invocation to the registered receivers.
This pattern is basically an EJB adaptation of the *Observer* pattern described in [GHJV94].

Patterns for Component Internal Structures

Quick Access Table

if...	use...
... Entity Beans contain persistence code and business logic and thus are hard to test	DATA ACCESS OBJECT: The persistence code is implemented in a separate class
... testing the business logic of your Entity Beans is hard because new versions must be deployed before testing	WRAPPED BUSINESS OBJECT: The business logic is implemented in a separate class
... deployment errors occur because methods in the Remote Interface are not implemented in the implementation	BUSINESS LOGIC INTERFACE: An interface is created that defines all business logic methods.
... persistent objects of finer granularity than Entity Beans must be defined	DEPENDENT OBJECT: The fine grained objects are implemented as parts of an Entity Bean

Data Access Object

As you use Bean Managed Persistence your Entity Beans are a mix of code for business logic and system level logic. Thus your Entity Beans are very complex as different concerns are mixed within its implementation. Therefore you are not able to develop and test your persistence code separately from your business logic.

Take a look at a typical Entity Bean using a relational database as its persistent store. In your *ejbStore()* / *ejbLoad()* / etc. operations, you will repeatedly use JDBC code to access the database. This includes connection and statement management and, what is more important, it contains SQL statements. This SQL statements might even have to be created dynamically what results in a lot of complicated String handling code .

Additional problems will arise, if you are forced to change your database vendor. Although SQL is standardized there remain little incompatibilities between different database products. Therefore you will have to change the SQL statements all through your Entity Beans.

Another problem you face is that if you want to test or debug the database specific code of your Entity Beans you have to deploy you beans to an application server. This makes testing complicated and for debugging you need to have tools supporting debugging in an application server.

Therefore:

Add an additional class that holds all logic needed to handle persistence for a special Entity Bean. This is a so-called DATA ACCESS OBJECT. Then your Entity Beans call the DATA ACCESS OBJECTS within their lifecycle callback methods to handle persistence.

❊❊❊

General

Your Data Access Object has to get and set the members of your Entity Beans. You have different ways to implement this. The simplest way is to let the Data Access Object directly access the members of the Entity Bean. This is a very tight coupling between both and makes separated testing of the Data Access Object difficult at least. A better way is to use Value Objects to exchange the data between the Entity Bean and the Data Access Object. The problem with this variant is that through frequent generation and destruction of these Value Objects you generate higher load on the application server. When you have very frequent access to the Data Access Object using Value Objects to exchange data this can result in bad performance.

Advantages

Using Data Access Objects has several advantages over handling Bean Managed Persistence from within the Entity Beans directly. First your Entity Beans become less complex as they only contain the business logic and delegate the persistence operations.

You can develop and test your persistence code separately from the rest of your Entity Bean as you follow the Separation of Concerns principle.

Also if you use Data Access Objects you get a more flexible architecture. As the requirements of your application change, you might need to switch from Container Managed Persistence to Bean Managed Persistence to a Type Manager or the other way around. If you use Data Access Objects switching will become much easier because your Data Access Objects can be used by both an Entity Beans with Bean Managed Persistence and a Type Manager. Also switching between Bean Managed Persistence and Container Managed Persistence is easier, because you clearly separated your database code from the rest of your bean.

Drawbacks

As you introduce a new class for handling persistence have a more complex Component Implementation. This leads to additional efforts for developing and maintaining your Entity Beans.

You also introduce a new level of indirection, which gives a small performance degradation. Nevertheless this indirection is minimal compared to the interprocess communication between client and server, the interception stuff the container does like checking security and transactional settings and the actual database access. So it usually does not lead to a visible performance degradation.

Variants

If you use Factorys [GHJV94] to get the right Data Access Object you achieve loose coupling between your Entity Bean and your Data Access Object. So you can easily change the code for persistence if you need to change your persistence technology or your database schema.

Also consider a framework based approach to generate SQL Statements to get database independency.

Related Patterns

The Wrapped Business Object pattern does the same things for business logic that Data Access Objects do for persistence code. Applying both patterns degrades an Entity Bean to be only an "infrastructure adapter" for business logic and persistence code.

Dependent Object

Components which represent business entities (i.e. Entity Beans or Type Managers) are relatively large-grained artifacts. Usually these entities have a more fine grained internal structure – some attributes form a semantic group, and sometimes these groups can even be required more than one time. However, these groups are no independent entities themselves.

Again take a *Person* component as an example. A *Person* usually consists of the standard attributes like *name, first name, middle initial* and *date of birth*.

However, in addition you usually want to store other information like addresses, telephone numbers or email/web addresses. An address for example, consists of several attributes itself, such as street, zip code, city and country. From the perspective of the *Person* COMPONENT, these attributes form a semantic group and belong together. The same is true e.g. for area code/phone number.

Another consideration is that a *Person* might not only have one address or one phone number. In most realistic applications, several addresses are required for a *Person* – maybe a home address, a work address, etc. Usually you don't know in advance which and how many addresses are required.

Making things like addresses and phone numbers entities in their own right is not acceptable for two reasons: First, they are tightly coupled to their "hosting" entity – their is no use in an address or a phone number without being attached to a real entity. The second reason is performance. Creating separate entities for each group of attributes would incur too much overhead in the server.

Therefore:

Partition the state of an entity into several dependent objects. Each dependent object represents a semantic group of values that belong to the entity. The lifecycle of a dependent object is identical to the lifecycle of its hosting entity – they cannot exist on their own (composition) and they have no own identity and are usually immutable. Dependent objects can have zero, one or many instances, depending on the requirements of the application.

❊❊❊

Advantages: As you use Dependent Object you have the advantage of keeping the structure of you COMPONENT better organized. If you need to store certain semantic groups of attributes more than once per entity, dependent objects are the only reasonable way to achieve this.

Drawbacks and Caveats: Please note, that dependent objects have no logical identity, because they are only part of the state of their hosting COMPONENT. This has some subtle consequences: While it is easy to, say, add a new address to the list of addresses of a COMPONENT (you simply pass in a new address object) it is much harder to remove one, because you cannot easily identify the "one". To overcome this problem, there are several ways:

- Either, a dependent object's "identity" is defined by the set of the values of its attributes. Thus, if you want to remove the address *High Street 10, 89520, Heidenheim, Germany* you pass to the remove *operation()* a dependent object with these parameters – if all are equal, the object is deleted from the COMPONENT'S state.
  Because dependent objects have no identity, and because as a consequence, they are usually identified by the set of the values of its attributes, you should make sure that dependent objects cannot be modified once they are created – they should be immutable. If modifications would be possible, there would be no way of identifying a specific dependent object, because its identity changes with the changes to its attributes. So, if you want to modify e.g. the zip code of one of the addresses, you have to create a new dependent object with the modified zip code, remove the old one, and add the new one.
- Or, to facilitate equality comparisons, you can also use IDs which are invisible to the client. They are created by the server when the new instance is created.
- Alternatively you can provide setter and getter operations for the whole list of dependent objects. A client will then retrieve the list of addresses, modify it as he likes, and write back the whole list.

Which way is best depends on the use cases – if usually all instances are required but modifications happen seldom, you can use the list variant. The list variant has some drawbacks, though: The client who downloads the whole list "locks" all numbers – race conditions can occur, because the Entity does not support long transactions. In Java, because no generic programming is available, the list interface employs weak typing, thus the Entity must check the newly set list that only correct element types are provided.

Variants: Dependent objects can, but need not be visible in the interface of the COMPONENT. If they are visible, they act as VALUE OBJECTS and clients can directly create such dependent/value objects and pass them in as parameters, or receive them as return values. To simplify client programming, provide factory operations to create "empty" instances of these dependent objects. The downside is that you "publish" the COMPONENT's internal data structure, which makes changes to this internal structure harder – you have to adapt the clients, too. If you choose not to make the dependent objects visible, the application can either use separate VALUE OBJECT classes, or use BULK SETTERS to add or remove the attributes contained in dependent objects.

Additional Information: Use of dependent objects is simplified significantly, if the CONTAINER supports automatic persistence for entities consisting of dependent objects. If this is not the case, you should create your own "framework" for managing dependent objects' persistence.

Please note also that dependent objects are not the same as relationships among entities. Entity relationship usually implies that the related entities are entities – which means that they have their own identity. And entity relationships usually incur some more performance overhead than dependent objects. Of course, there are situations when it is not easy to decide how do design the system, for example, if you use the same addresses for customer, employee, and consultant entities, than it might make sense to manage addresses as separate entities. A RELATIONSHIP SERVICE can provide help here.

Related Patterns: This pattern is similar to VALUE OBJECT. The difference between DEPENDENT OBJECT and VALUE OBJECT is that VALUE OBJECT has been introduced because of the observation of frequent usage of groups of parameters by clients and is primarily a way to optimize performance by minimizing the required number of network hops. DEPENDENT OBJECTS are used to structure the internal state of a COMPONENT.

DEPENDENT OBJECTS are usually good candidates to be subject to LAZY STATE or DEFERRED STORE, because they can be rather big and are not used in every situation when the Entity is accessed.

Note that the current draft of the EJB 2.0 specification contains an implementation of this pattern. However, the specification is still in the standardization process and especially this part is not stable yet.

Wrapped Business Object

You have implemented complex business logic in your beans. Because your business logic is very complex your beans are very complex. Developing, testing or debugging your business logic is hard, because the bean has to be deployed and executes in the CONTAINER.

Imagine the implementation of a *Customer*. Besides the attributes there is a method that tests whether the *Customer* is credit-worthy for a certain amount of money. The implementation of this method is quite complex because the credit-worthiness of a customer does not solely depend on his attributes but also some checks in a legacy system need to be done – and then a complex calculation follows. Because this is quite error prone it must be extensively tested and for convenience this should be done outside the application server.

Another example is a service that handles matrix multiplication. This service is relatively complex to implement. If you implement it within a Stateless Session Bean you have problems when you need to test and debug the implementation. You would need a debugger that supports debugging within the container if you want to debug the deployed bean. Although such remote debuggers are available, normal debuggers are easier to use.

Therefore:

Add an additional class that holds the business logic, a so called Business Object. Use your bean to

delegate calls for business operations to the Business Object. So your bean is only a wrapper. For testing and debugging, use the Business Object class directly.

❊❊❊

General

Advantages

Using WRAPPED BUSINESS OBJECT separates business logic from EIB-specific requirements. Therefore it is possible to implement and test the business code independently. Because you separated your business code from the code that deals with the CONTAINER, you can also use the business logic in other parts of your application, for example in a fat client.

Drawbacks

A drawback is that you have to deal with additional classes and/or interfaces for every COMPONENT, so you may have a lot of more classes in your system. So you should only use WRAPPED BUSINESS OBJECT if your business logic is complex and needs to be tested and/or debugged independently from the CONTAINER.

Variants

There is a specific problem with container managed persistence: you can usually only use members of the bean as persistent attributes. So you have to implement the logic in your business class and have its members in the bean itself. This leads to tight coupling between the business class and the bean, what abolishes the advantages of WRAPPED BUSINESS OBJECT. So a solution to this problem is having the members both in the bean and in the WRAPPED BUSINESS OBJECT. Before the WRAPPED BUSINESS OBJECT will be used the bean has to fill the members of the WRAPPED BUSINESS OBJECT with its own data.

Another implementation variation is to derive your beans implementation from the Business Object class. This also solves the problems with container managed persistence for most containers. In addition, you don't have to implement any code to delegate calls from your bean to the Business Logic class.

Related Patterns

Note the difference to the STRATEGY pattern: the goal of STRATEGY is to make some aspect of the overall business logic exchangeable. With WRAPPED BUSINESS OBJECT the complete business logic will be moved to a separate class to facilitate testing and debugging. Motivation of WRAPPED BUSINESS OBJECT is not to make the business logic exchangeable but to clean up the structure of the bean and make development easier. STRATEGIES can still be used inside a WRAPPED BUSINESS OBJECT.

If you implement WRAPPED BUSINESS OBJECT consider using BUSINESS LOGIC INTERFACE to guarantee that the bean and the WRAPPED BUSINESS LOGIC implement the same interface.

DATA ACCESS OBJECT is a similar pattern for separating the database access code from the component. If you use WRAPPED BUSINESS LOGIC and DATA ACCESS OBJECT, your COMPONENT only deals with system specific stuff and acts as FACADE for the business logic. This SEPARATION OF CONCERNS gives you high flexibility for changing technologies and makes development easier as the structure and purpose of your classes is simpler and therefore easier to understand.

Business Logic Interface

The COMPONENT INTERFACE is not technically related to the implementation of the COMPONENT. The means that error in the implementation of the COMPONENT INTERFACE are not found until the Component is deployed or verified.

A simple error in the implementation of the COMPONENT INTERFACE such as a misspelled method name can cause the deployment to fail. Usually this type of error if found by the compiler. But for Enterprise Java

Beans the implementation can not directly implement the COMPONENT INTERFACE. The reason is that the Component Interface includes methods that can only be implemented by the Container such as the methods that return a Handle to the component or remove the Component. Still the business methods defined in the COMPONENT INTERFACE must be implemented by the Component. Because this is neither enforced nor checked by the compiler errors in this field might be detected quite late.

Therefore:

Provide an interface that contains only the business methods that must be implemented in the implementation and declared in the COMPONENT INTERFACE.

❊❊❊

General

This ensures that the business methods are really implemented as declared. There are some minor problems. For example in the Component Interface the methods must be declared to throw RemoteException. However, the implementation is not allowed to throw these exception or declare them. So despite the fact that the methods must now be implemented to successfully compile the application the RemoteExceptions can still be declared or thrown and this error again will not be detected until the application is verified or deployed. Note that these errors are violations of the Specification but most servers still accept Beans with these errors.

Drawbacks

Some EJB implementations might have problems with the deployment of Beans that use this pattern.

References

This pattern is also explained in [MH00]. More details about this pattern especially concerning the Java type system can be found in [Wo00].

Patterns for Building Large Systems

Quick Access Table

if...	use...
... components should not know which other components are responsible for specific tasks to further decouple them	REQUEST HUB: A central component forwards a request to the responsible component
... components must have some installation code that sets up the environment	ADMINISTRATABLE COMPONENT: Add an additional interface that allows the installation code to be called.
... some components of the system form a group but no formal grouping is present	BUSINESS COMPONENT: Provide an additional component as a GoF FASCADE
... a functional aspect that is not covered by the Conatiner must be enforced on all components	ROLL YOU OWN INTERCEPTION: Provide code that is executed before and after each method call
... a group of components needs the same configuration parameters	CONFIGURATION SERVICE: Provide an additional component that other components can use to access the configuration parameters

Administratable Component

In addition for providing the business functionality for the clients, a COMPONENT usually needs to perform some administrative, setup or diagnostics work, often directly after COMPONENT INSTALLATION. If you want to reuse a (BUSINESS) COMPONENT as a complete, reusable package, the COMPONENT must be able to achieve this task itself – however, clients should not be able to access this functionality.

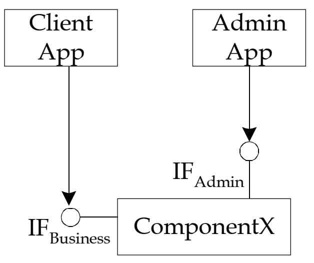
A large BUSINESS COMPONENT for managing the contacts of a company (people, customers, suppliers, etc.) is made up of a set of collaborating COMPONENTS. When they are installed, they rely on some external resources, such as the schema in the database. You do not want to require the users of the COMPONENT to create the schema manually, instead your component should be able to do it itself.

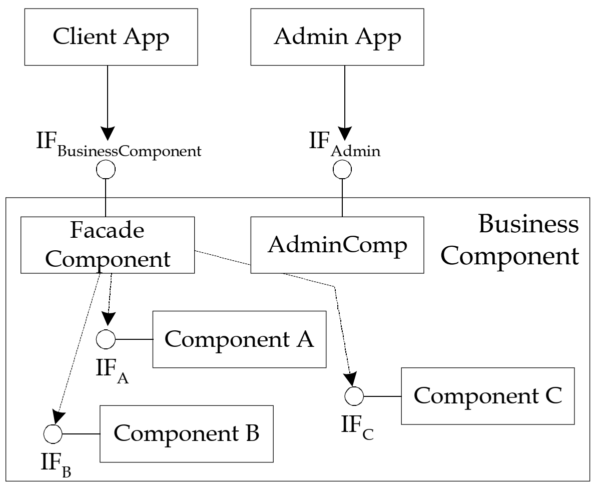
The same is true for some kind of "installation test" which checks, that the COMPONENTS have been installed correctly, all its resources are available, and generally, no unexpected error occurs.

Later, when the system is running you might want to access some diagnostic information about the COMPONENT(S), such as the number of errors that occurred (and a list of them), etc.

Therefore:

Implement administration, setup and test functionality as part of your (BUSINESS) COMPONENT. Provide access to this functionality only for specific administrators, not for all clients. Make sure that the necessary operations are not visible in the business interface.





❊❊❊

Variants: There are two ways how to implement this pattern because the implementation is different for normal EJBs and BUSINESS COMPONENTS built with EJBs.

For normal EJBs, you have to make sure that there are two COMPONENT INTERFACES for the COMPONENT. One provides the business functionality, and the other provides the administration operations. Define the interfaces separately, and create a common interface (extending the two other ones) which serves as the remote interface for the bean. Your clients (should) only know about the business interface. In addition, use ANNOTATIONS to define security properties, that allow only some *administrator* role to access the administrative operations. Another way of implementing this Pattern is to use MULTIPLE INTERFACES: The component has one interface for administrational purposes and one "real" business interface.

In BUSINESS COMPONENTS, which consist of a group of EJBs, provide (one or more) additional EJB that take care of the administrative functions. Once again, use security settings in the ANNOTATIONS to prevent normal clients from accessing administrative beans. When using a SERVICE COMPONENT FACADE for BUSINESS COMPONENTS, make sure these operations are not part of this interface – the administration EJB must be lookup up separately.

It is useful to include a small client application with an ADMINISTRATABLE COMPONENT, which provides a GUI to access the administration features.

In CORBA's Component model, the operations defined directly on the COMPONENT (i.e. not on a facet) are reserved for exactly these administration purposes.

Business Component

It is often hard (and sometimes conceptually impossible) to "press" the complete functionality for a requirement into one EJB . Thus, you end up with a set of EJBs which are always used together as a group. However, there is no "formal" grouping for these COMPONENTS, and clients have a hard time because they have to operate on many instead of one COMPONENT.

A medical information system has a "subsystem" for managing operations. It has to keep track of the diagnoses and measurements done during an operation (such as heartbeat, breathing frequency), it has to record the materials used during the operation, it needs to store the doctors and assistants working on the patient, the complications, etc.

In addition, it has to provide overviews (lists) of all operations with specific personnel, or notify other assistants when certain conditions occur.

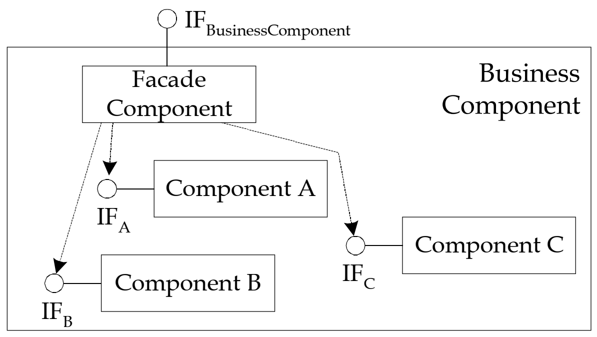
It should be obvious that it does not make sense to put this into one EJB , although the resulting set of

COMPONENTS belong together and will usually not be useful in isolation.

A client application has to know which of the COMPONENTS does what, and has to deal with all of them separately, depending on its use-case. A change to one of these COMPONENTS usually requires changing all clients.

Therefore:

Use an abstraction called a BUSINESS COMPONENT which consists of several EJBs internally. Distribute and release the COMPONENTS always as one "subsystem". Provide a *Facade* COMPONENT [GHJV94] which servers as the single access point to the whole business component, simplifying client access. This facade might use WEAKLY TYPED INTERFACES to simplify reuse and integration.



***

Advantages: This pattern provides a higher level of granularity for your COMPONENTS. This simplifies maintenance, versioning, and distribution.

Drawbacks and Caveats: However, as a drawback, some of the mechanics that are available for EJBs cannot be directly applied to BUSINESS COMPONENTS, because the CONTAINER does not know anything about this higher level abstraction.

Variants: The *Facade* COMPONENT can either be strict, hiding all the other components from the client (which requires data exchange to be done by VALUE OBJECTS, because entity Beans cannot be exposed). The facade then delegates all invocations to the respective Components. The alternative is to use a lazy FACADE, which means that the facade merely plays the role of a *Factory* for the other COMPONENTS, allowing clients direct access to all the component in the business component. Using the FACADE Component has all the consequences that are described in the *Facade* pattern in the GoF book [GHJV94].

Usually a business component supports complex processes that involve operations on several internal components as part of the process. Using the *Facade* COMPONENT to implement these processes has two advantages: First, performance is improved because the number of required network hops is reduced, and second, the *Facade* acts as a SESSION BEAN FACADE, relieving the clients from taking care of the correct workflow and transactional integrity.

If a strict facade is used, you can further abstract communication and reduce (formal) dependencies by using WEAKLY TYPED INTERFACES for the FACADE. Business Components will then only exchange "request objects" with each other. Further decoupling can be achieved by using a REQUEST HUB. To simplify consistent configuration of the set of internal EJBs, a CONFIGUATION SERVICE is helpful.

Related Patterns: Note that this pattern is not the same as SESSION BEAN FACADE. Although both use a session bean to abstract communication with a set of COMPONENTS, SESSION BEAN FACADE is used to enforce transactional integrity and improve performance, whereas a business component is a higher level design artifact.

Configuration Service

If you have several related components, for example in a Business Component, they usually have some aspects of their configuration in common. These aspects can be technical or functional. Specifying all the Configuration Parameters in all the Annotations is inefficient and error prone.

A a Business Component for address consists of many Components. Because you do not want to limit the chances for reuse of the Components, you make the regular expression that validates ZIP codes configurable. Such validations of ZIP codes are performed by many different Components in the Business Component. However, when the Business Component is deployed as part of an application, there is usually only one ZIP code verification expression in use – and this expression is used by all the Components inside the Business Component.

To make sure your application works correctly, you do not want to configure this regular expression for each of the Components separately.

Another example is internationalization. The text messages for exceptions must change according to the selected language, but all Components should use the same texts (and the same language!).

Therefore:

Provide a central configuration service. The participating Components access it (usually during initialization) to retrieve their own configuration. It is also possible to let clients access the service, if necessary.

❊❊❊

Advantages: This pattern reliefs you from doing the same configuration again and again. It is especially useful if some part of the configuration can be inferred from other parts, reducing the configuration work even further.

Variants: There are two basic ways how the pattern can be implemented:

- The configuration service can be a real Component. This imposes some performance overhead but this overhead is not very problematic because the access to the configuration service usually only occurs when the client Component is really physically instantiated (i.e. usually when the pool is created).
  On the plus side, this allows clients to access the configuration information too, providing a kind of centralized reflection or configuration repository. I.e. also parts of the GUI can be adapted to the configuration.
- The other possibility is to use a simple Java class. Each Component creates an instance, and accesses a central configuration file (through a Managed Resource's factory) to read its configuration. This is a more lightweight solution, but the configuration is not easily made accessible for the client.

You can even make the configuration depend on logical entity, or another part of the runtime state. This allows you, for example, to have American or German addresses to be managed by one Component in one application. Of course because of the Distinction of Identities, the configuration must be redone for each change of logical identity – it's not enough to do it when the physical instance is created.

Request Hub

When a system is built from a set of collaborating BUSINESS COMPONENTS, coupling can be reduced by using a WEAKLY TYPED INTERFACE. However, a COMPONENT still needs to know, which other COMPONENT is responsible for fulfilling a specific task. In addition, these components depend on each other's interfaces, including operation parameters, VALUE OBJECT types, etc. This compromises independent evolution of systems and BUSINESS COMPONENTS.

Imagine a e-business application. It uses several BUSINESS COMPONENT: A personalization and cross-selling engine, a catalogue engine, an order management and several warehouse subsystems, one for each branch of the business. They have to collaborate in order to fulfill the application's purpose. They all use XML-based WEAKLY TYPED INTERFACES, because all subsystems have been developed by different third party vendors.

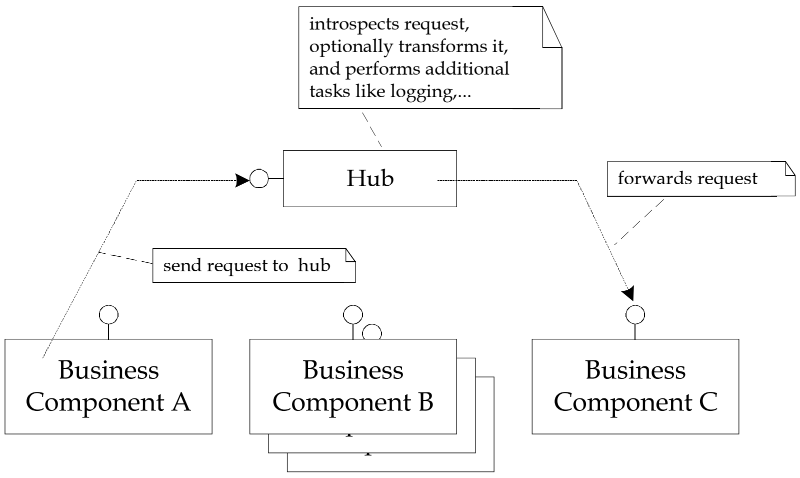
The controlling component (could be calls the workflow engine) has to work with all these components. Although all have an XML based interface, the data structures (DTDs) they use are different. They contain the same content, but their structure is not standardized. So you need a way to adapt the data formats in the communication among the BUSINESS COMPONENTS.

Also, when new releases of these software packages are introduced into the system, you need to have a way how one interface can be modified without requiring the adaptation of all your clients – partly, because they come from different vendors.

As a third aspect, if several BUSINESS COMPONENTS provide overlapping functionality, you might want to make another COMPONENT responsible for a specific task, without changing all the clients.

Therefore:

Provide a central COMPONENT in the system, which receives every request and forwards it to the correct receiver. In addition, this request hub can adapt the parameters, change data types, provide default parameters for modified interfaces, etc. It can also be used to mask different communication protocols.



***

Advantages: As mentioned in the introduction example, a request hub is particularly well suited to "glue together" a set of 3rd party COMPONENTS, because it can be used as a centralized adapter for all the COMPONENTS. This is especially useful if some of these COMPONENT use a non-EJB communication strategy, such as SOAP, or low-level sockets.

Drawbacks and Caveats: The hub is a central system component. To do its job, it needs to know which part of the system is responsible for which task – when responsibilities in the system change, the hub must be updated.

Additional Information: It is important to understand that the hub can, but should not be used as a replacement for a CONTAINER, who should take care of the technical concerns like load-balancing, transactions, fail over and security. Implementing these aspects is not trivial (it's also not trivial with a request hub), this was the reason why a CONTAINER has been introduced into a COMPONENT system. However, it is possible to implement technical concerns in a request hub: For example centralized logging or more sophisticated security policies can be implemented here if the CONTAINER does not support them. A simple form of application level load-balancing is also easily possible by forwarding requests to different COMPONENTS upon invocation.

Note that the hub is not necessarily a single point of failure. It can easily be replicated on different machines because it is stateless – you just have to make sure that the different hubs are all configured the same way.

Variants: If no too complex request transformations are necessary, the hub can be a generic COMPONENT, because WEAKLY TYPED INTERFACES are usually reflective, so the routing policies can be configured and do not need to be programmed. If this is not so, consider using *Strategy* to make it flexible enough.

In addition to the above-mentioned responsibilities, the hub can take care of pseudo-technical concerns, like centralized logging, asynchronous request forwarding, storage of messages if a component is not available, etc. It provides a central location where these concerns can be configured, e.g. using policies, etc.

Forwarding of requests can happen either synchronously, by just invoking operations, or it can be based on an asynchronous communications mode, e.g. using JMS.

Related Patterns: Because every request travels through the request hub, it is a good place to hook in INTERCEPTORS, as such, a request hub can be an alternative to ROLLING YOUR OWN INTERCEPTION.
This is basically an implementation of the *Mediator* or the *Broker* patterns.

Roll-your-own Interception

When creating large systems, you will often come across some requirements which you want to enforce in the project, but cannot be implemented in the CONTAINER. CONTAINERS usually do not support these features, because they are not technical in the sense of the CONTAINER, but they are also not functional from the perspective of your developers.
Hard-coding these requirements in the COMPONENTS is not acceptable – especially if you want to be able to enforce certain policies in the project, or if you want to be able to retrofit some of these requirements after the COMPONENTS have been implemented.

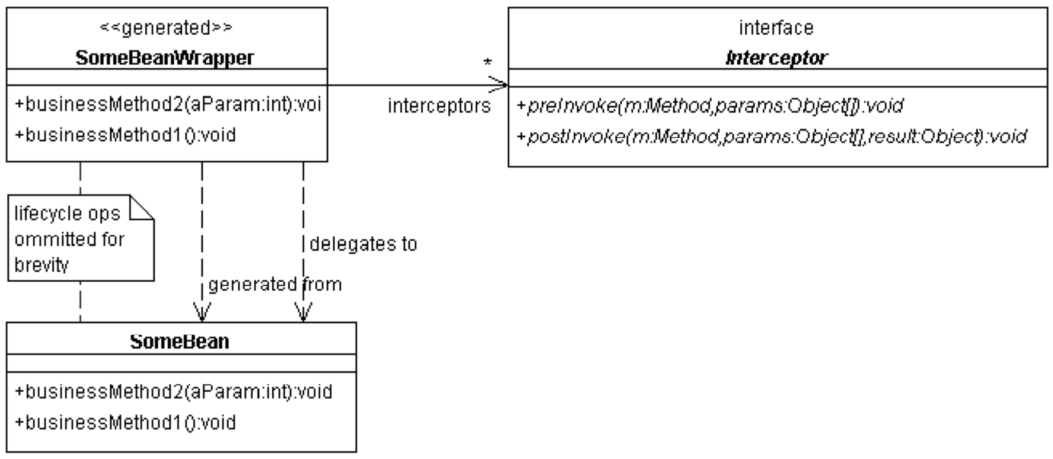
A typical example is resource access decision. The EJB standard only allows you to "hard-code" which roles of users can access a certain operation. But you cannot code things like "Any Entity can only be accessed if the Principal is in the same team as the person who created the Entity." For the COMPONENT implementor it is a daunting and error-prone task to incorporate a corresponding check into each operation and throw an exception if necessary.

If you want to retrofit these checks after the COMPONENTS have been implemented, you would have to change each COMPONENT in turn, which is usually unacceptable.

Another typical case for this problem is performance probing. On a website you might want to log which operation is called at what time and by whom, for example to enable billing or just for statistics; or you are interested in how long the execution of an operation takes in order to measure the performance of your website.

Therefore:

Create your own INTERCEPTION interface. Use code-generation to create a wrapper bean implementation which delegates pre- and post operations for each business operation to one or many interceptors. Use Java's reflection to marshal the information about the operation called. The wrapper then forwards the invocation to the real implementation class.



<<generated>>
SomeBeanWrapper
+businessMethod2(aParam:int):voi
+businessMethod1():void
interface
Interceptor
+preInvoke(m:Method,params:Object[]):void
+postInvoke(m:Method,params:Object[],result:Object):void
*
interceptors
lifecycle ops ommitted for brevity
delegates to
generated from
SomeBean
+businessMethod2(aParam:int):void
+businessMethod1():void

❊❊❊

Advantages: This pattern allows you to take any COMPONENT implementation created by one of your "ordinary" COMPONENT programmers and plug in stuff created by infrastructure people.

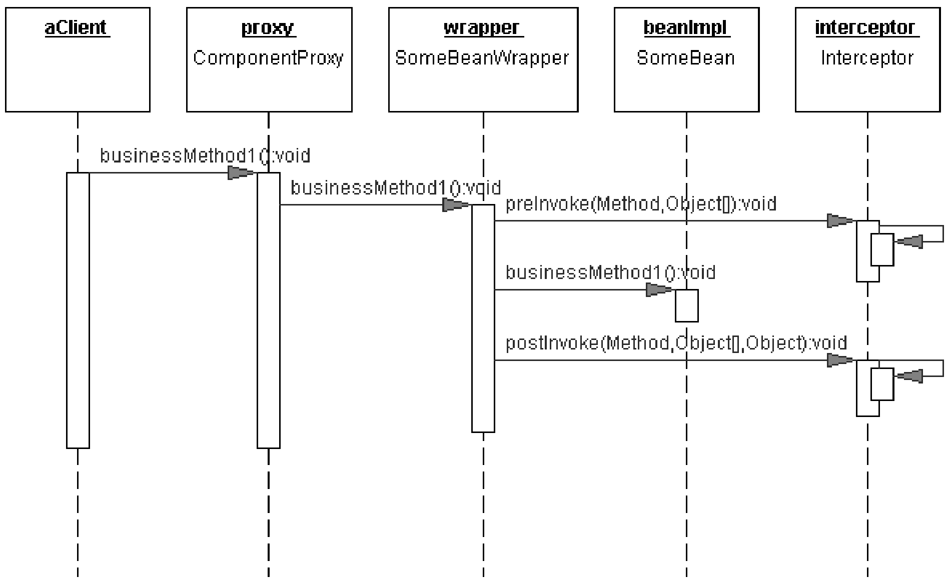
Drawbacks and Caveats: Code generation is always a bit problematic because of the sometimes awkward integration into the development environment. However, in the case describes here it is not so much of a problem, because COMPONENT INSTALLATION requires code generation anyway, and the generated code is simple and well structured.

Additional Information: A special code generator inspects the *SomeBean* implementation class and creates a wrapper class that implements each operation by executing the following steps:

1. invoke *preInvoke()* on each interceptor available
2. invoke the original business operation on the original bean implementation
3. invoke *postInvoke()* on each interceptor available

This process is illustrated in the following sequence diagram:

Upon deployment, you have to specify the generated wrapper class as the implementation class in the ANNOTATIONS, not the original bean implementation, because method invocations must arrive at the wrapper to allow it to invoke the interceptors. Because it has the same operations as the normal implementation, this is always possible.



To allow deployers to flexibly add or remove interceptor instances, the required interceptors must be specified upon deployment in the CONFIGURATION PARAMETERS. The wrapper has to check these parameters when it is instantiated.

Related Patterns: Because usually you will use the same interceptors for many COMPONENTS in your BUSINESS COMPONENT, a CONFIGURATION SERVICE can be a great help here.

Persistence Optimization Patterns

Quick Access Table

if...	use...
... old data must be kept instead of deleted e.g. for legal reasons	HISTORIZED STATE: keep the old data and introduce a deleted flag and timestamp.
... access to the database or to the data in the legacy system is too slow	MIDDLE TIER CACHE: The data is cached in the Application Server
... updates in the database happen frequently but seldom data is really changed	Deferred Store: Store the data only if it has been really changed. Use a flag to mark changed data.
... complete Entity Beans are loaded into the Application Server but only parts are usually used	Lazy State: Load the the state only when it is needed.

Historized State

For legal or other reasons you want to keep old data. However, the Enterprise Java Beans specification requires you to delete it once ejbRemove() is called and there is no archive mechanism built in.

Usually ENTITY COMPONENTS are deleted or data is changed without keeping the old data. So a change will just overwrite the previous data. But this is often unacceptable for legal reasons. The law might require that old data is archived instead of deleted so that old data can be reconstructed. In this case it is often enough to store the old data so it can be reproduced by manual SQL queries (historization). But it might make sense to store old version to make sure that user errors can be undone. In that case the old data must be accessible to the application and the user (versioning).

Therefore:

Implement the ENTITY COMPONENT methods in a way that the data is kept instead of deleted or just changed. Introduce a timestamp or version in the database. Mark database entries as deleted instead of actually deleting them.

❊❊❊

General

Basically the idea is to implement the operations required by the LIFECYCLE CALLBACK of Entity Beans in a way that keeps the old data in the database. This means for example that instead of an SQL delete just an SQL update marks the database entry as deleted. If the data should be update the SQL update is replaced with an SQL insert with a new version number. SQL select to retrieve the database entry must use the database entry with the highest version number and they must not return entries that are marked as deleted.

An interesting problem is the primary key. Usually the primary key can largely stay unchanged i.e. if the primary key without historization was a String it will stay a String and the timestamp will just be kept internally. So if you try to access the Entity Bean with a certain primary key you are presented with the data row with this primary key and the highest version number or timestamp. Note that the primary key for the Entity Beans can not be the primary key in the database: it is not unique because several entries with the same primary key and different timestamps might exist.

If you want to make the different versions of the Entity Beans accessible to clients you must make the primary key contain the timestamp as well as the original primary key. Otherwise there is no way to distinguish between different versions of the Entity Bean. This primary key usually contains a timestamp and the old primary key e.g. a customer name or customer number. This leads to the problem that usually not a specific timestamp is of interest but rather the version that was valid at a certain point in time. Imagine you want to see the address of a customer as it was on March, 31st 2000. You are not interested whether the timestamp for this entry is from January, 2nd or March, 30st. It is just of interest that the version was valid on March, 31st. So find methods must be defined that allow the user to look up the version of the Bean that was valid at a certain date. The access for the versioned data should be read-only usually because it is archived data and should just be shown to the user. Changes of this data make usually no sense and are hard to handle. This means neither *create()* method nor *ejbStore()* or *set...()* methods should be implemented.

Related Patterns

To make sure that new version numbers are only created when the data actually changes you should use the DEFERRED STORE pattern.

Lazy State

You notice that a lot of time is spent for loading the state of Entity Beans. They represent complex business entities with lots of data. The data is loaded in one large database operation ("big inhale") but often only parts of the data are needed.

Entity Beans represent complete business entities with all their data. If you use Container Managed Persistence or you make a naïve implementation using Bean Managed Persistence this means that for each entity all data is loaded form the database and stored after the work on the data is done. This is acceptable for simple and small Entity Beans. In fact in this case it is the best choice because just one database operation is used and this keeps the overhead small. However, for large and complex data this operation might take a lot of time. If only parts of the data are usually used this is a waste of time as well as memory. Splitting the Entity Beans in multiple small Entity Beans is not a good idea because components should be coarse-grained. Also this would cause additional complexity and lead to a logical entity being split into multiple technical entities which in turn makes it hard for a client to use the component.

Therefore:

Load the data of the Entity Bean when it is really used. This can be done in the accessor operations for example. Because you will develop the database code yourself you need to use Bean Managed Persistence.

***

General

This pattern is also widely used in object-oriented databases systems (OODBMS). They usually load just one object without any referenced objects. Only if references to other objects are resolved, the referenced objects are loaded into memory. OODBMS try to further improve the performance by clustering together several objects that are usually accessed en bloc and then one such block can be loaded at once so that the referenced objects are already in memory when they are accessed. Using all these advanced techniques LAZY STATE performs well even if only small objects are referenced.

Things are different when LAZY STATE is used with EJBs. Every piece is loaded with its own database operation. So a performance benefit can only be achieved if the loading of large pieces of data is deferred. Otherwise the overhead of the database operation will be too big to note any performance improvement.

Related Patterns

An alternative to this pattern might be to use a TYPE MANAGER: The TYPE MANAGER loads all data directly from the database an thus never loads any unneeded data. However, this also means that each access to the data causes a database access and while with LAZY STATE the data is kept inside the Entity Bean and thus it is cached.

Middle Tier Cache

If an application uses business entities heavily, slow database access can become a serious bottleneck. Slow databases are often found in the case of heavily used legacy backend systems, usually, because many different applications work on the database concurrently. In many cases, however, an application only needs access to a certain subset of the data in the legacy database.

Consider an application that manages stock exchange data. A large backend database has been storing the history of each stock symbol over the last 20 years. The online trading application feeds data into the database, and many other systems also access the database concurrently.

Now imagine you need to build a web-based information system for the stock symbols. The slow database performance is unacceptable for a web-based service, and handling the heavy user load at peak times would further slow down the database, raising problems for the trading applications.

In cases such as these data access is typically read-only, and only a subset of the data is accessed regularly: The stock symbols of the current day.

Therefore:

Install a middle tier cache into your application, which pre-loads data from the legacy system of which you know that it will be needed often. Your entities persist only to this cache – they know nothing of the underlying backend database. It is the responsibility of the cache to decide which data is cached, and when to write (potential) changes back to the legacy database.

❊❊❊

Additional Information: The middle tier cache is responsible for keeping the entities synchronized with the database. The pattern is most useful if the application which uses the cache only reads the data, or if it is the only application modifying a specific set of data items.

In the first case, cache is used primarily as a read cache, propagating changes directly to the backend. In our stock trading example, this would be the best solution. However, there are also applications of the second case: Consider a situation where an application is used to gather data all over the day, whereas only the last item (or some average) has to be stored permanently in the backend. Then, the cache would collect data over the day and store it into the legacy database only once a day.

In case of a cache miss – i.e. the requested data is not available – the cache must query the backend to retrieve the data. However, it need not necessarily cache this data for further access if it decides that the data will probably never be used again.

A middle tier cache is not a completely generic artifact. To be useful, it at least needs to know which data to cache, and how long. Because this usually depends on the application requirements and the usage patterns, this cannot be done completely automatic.

It is important to understand that a middle tier cache can, but should not be used to overcome weaknesses in the CONTAINER's caching and POOLING strategies. The middle tier cache can use information about the access patterns of the data to prefetch and cache data that will very likely be used. The CONTAINER does not have this information and thus will not be able to provide this kind of service.

Drawbacks and Caveats: In multiserver architectures the middle tier cache should exist in one instance only. Otherwise the different Middle Tier Caches have to synchronize their content.

Variants: A middle tier cache can be implemented using several different persistence technologies. In the simplest case, the data can be stored in memory, of course with the drawback that it is gone if the server dies. Another solution is to use another database of the same type as the backend (or at least a compatible one) and do the transfer using script programs and database command line tools. Another nice alternative is to use object-oriented database here, because the data model of entities (usually made up of DEPENDENT OBJECTS) can be persisted more or less directly.

Related Patterns: Note that this pattern works equally well with ENTITY COMPONENTS and TYPE MANAGERS. After all it is just another persistence layer for the entities. The interface to the middle tier cache can be implemented as a COMPONENT itself to facilitate programming. Alternatively in can be done as an EXTERNAL SERVICE, or by using direct (cache) database access.

Deferred Store

You notice that your database is updated frequently even though the Entity Beans are seldom changed. This leads to performance degeneration.

The state of an Entity Bean is frequently stored in the database, at least at the end of a transaction. But often the state of the Entity Beans did not change and thus time is wasted with database operations. The frequent calls are only made to synchronize multiple Entity Beans that represent the same data row: If two Entity Beans represent the same Customer and the data is loaded before each method call and stored after each method call, the Entity Beans appear are synchronized through the database. However, the Container has no means to detect whether the state was really changed. Thus a lot of unnecessary updates might happen.

Therefore:

Only store the state in the database if it is changed. Introduce an instance variable in the Entity Component to keep track of whether the state changed or not.

***

General

This technique is also often used by object-oriented database systems (OODBMS). If an object is changed it is marked "dirty" and only then it is stored in the database. Otherwise no storing is done. The problem is how the OODBMS notices when an object is changed. Some OODBMS require the developer to do it by hand, others do dirty tricks with the memory management or insert additional code into the application.

For the purposed pattern this must be done manually i.e. each time the value of an instance variable is changed a boolean instance variable must be set so that the entry is written back into the database. After it was written back or the state was read from the database the variable must be reset.

From a performance perspective this pattern should be used for every Entity Bean as the overhead for checking and setting the variable is negligible while the gain from using less database operations can be quite significant. The problem is to really set the variable each time the state of the ENTITY COMPONENT changes. If you make an error in this code a change might not be propagated to the database and thus be lost.

Variant

It is also possible to implement a finer grained modification control: If some of the instance variables contains most of the data it makes sense to implement a boolean variable for each of them to indicate changes and store only those instance variables that were changed. This is closely related to LAZY STATE: While LAZY STATE tries to avoid unnecessary loading of complex data DEFERRED STORE can in this case be used to avoid unnecessary storing of complex data.

Things get more complicated if you include more complex business logic in your ENTITY COMPONENTS. This opens more possibilities to change the state of the component without setting the variable. You can try to avoid this by using access methods instead of accessing the variables directly. To enforce this you can built a class that contains the instance variables as private variables and has public accessor methods for them. In the subclass that is the actual implementation of the ENTITY COMPONENT you can implement the business methods. As these can only access the instance variables through the accessor methods must be used.

Patterns for State Access

Quick Access Table

if...	use...
...often some attributes of an Entity Bean are modified together	Bulk Setter: Introduce an additional method that allows the modification with one method call
...often some attributes of an Entity Bean are read and modified at the same time	Value Object: Use one object to represent all accessed attributes
...often attributes of several Entity Beans are read and modified at the same time	Combined Value Object: Use one Value Object that contains the data from the different sources
...often some attributes of an Entity Bean are read and modified together. Which attribute are modified changes.	Hashtable Value Object: Store the attributes as key-value pairs in a Hashtable used as Value Object.
...often some attributes of an Entity Bean are read and modified at the same time. There are a lot of combinations of attributes that are accessed.	Value Object Creator: Provide a generic Factory method to create the different types of Value Objects.
... many datasets must be transferred from the server to the client e.g. to display them	Bulk Reader: Introduce an additional Session Bean that directly accesses the database and returns a set of Value Objects.
... many datasets are modified in the same way	Entity Bulk Modifications: Introdue an additional Session Bean that does the modification directly on the database.
... many datasets must be transferred from the server to the client one-by-one e.g. to modify them	Server Side Iterator: The data is transferred transparently in chunks.

Bulk Setter

Each call to an Entity Bean takes time because a remote invocation is necessary. This makes such a call expensive. In addition, every call to a bean will be intercepted and checked by the Container imposing even more overhead. Depending on the implementation of the Container, each call may even involve one or more database accesses for each operation to synchronize the state of the Entity Bean with the database. Thus, repeated invocations of setter operations on a bean are inefficient and should be avoided.

Imagine a *Person* bean which has setter operations like *setName()*, *setFirstName()*, *setDateOfBirth()*. Although it makes sense to be able to set each of these attributes on a one-by-one basis, it is often necessary to update all three of them at the same time. Especially if you have GUI dialogs you may often need to set group of attributes. Using the setters require three separate invocations, each traveling over the network and being intercepted by the Container.

Therefore:

Provide an additional operation, a so-called Bulk Setter. It takes a group of attributes as parameters that are often updated together. Whenever these attribute combination is updated, use the Bulk Setter instead of subsequent invocations of atomic setter operations.

❊❊❊

Advantages

Using this pattern will allow your application to scale better and make network overhead less of a problem. A side effect of this pattern is that if you group many setter operations together, the properties will be set within the same transaction. Nevertheless performance is the motivation for this pattern.

Variants

It is possible to remove the atomic set/get operations completely, when using this pattern, however, this is not always a good idea. It depends on the use cases – if you need to change only one attribute make atomic accessors available. If you only update groups of attributes (e.g. after submitting a form) then you can safely remove them partially or completely from the Component Interface. Bulk Setter calls also have the side effect that all parameters are updated in *one* operation, which usually means: during *one* transaction. Therefore, if you want to ensure that a set of attributes always updated atomically, use this patterns and remove the atomic setter operations.

If the atomic setter methods are also available, use them to implement your Bulk Setter operations. So you still modify your members at only one position of the code and have the possibility to add or change setting behavior locally. You don't need to make the atomic setter operations visible to the client. They might be bean internal operations, i.e. they do not need to be part of the Component Interface.

Related Patterns

If, in addition to updating, the reading process should also be made more efficient, or the parameters build a semantic group use Value Objects instead. An advantage of a Bulk Setter compared to Value Objects is that Bulk Setter are easier to implement because they don't add a new class to the application. Adding a simple operation adds much less complexity to your Component.

If the parameters form a semantic group and the lifecycle of the parameter set is linked to the lifecycle of the Entity Bean use Dependent Object. If you have many Bulk Setters or you often have to add new Bulk Setters and therefore often need to change the interface of your bean you might use Hashtable Value Object or Value Object Creator.

Value Object

Each call to an Entity Bean takes a relatively long time. This is because a remote invocation is necessary and expensive. Also for every call of a beans operation the Container has to check security and transactional settings. So many invocations of getter operations are therefore inefficient.

Imagine a *Person* bean which has getter operations for members like name, first name and date of birth. You often need to get all three values at the same time. One example is when you have to implement a User Interface. There you often have dialogs with groups of attributes being displayed together. You need to call many getter operations of the bean to retrieve all attributes displayed by the dialog. As you have multiple calls to your Container which all take some time to complete, displaying the dialog takes a long time.

Therefore:

Add an additional class which contains all the attributes you want to get at the same time. Also add a

Factory method to your bean that returns an instance of the new class that holds all the data. This is a so called Value Object. Whenever you need to retrieve all the information use the Factory method of your Entity Bean to get all the data with only one call. Of course Value Objects have to be transmitted by value.

❊❊❊

General

A Value Object has to be passed by value. To ensure this it must not extend the java.rmi.Remote interface. Just implement getter methods in your Value Object if you want to assure, no one changes the content of your object.

Advantages

Using this pattern will allow your application to scale better and make the network and the Containers interception overhead less of a problem. You can use Value Objects as parameter for other operations like setter or business operations, too. Therefore you get smaller and cleaner interfaces for your beans.

Drawbacks

A problem with using Value Objects is that you hold bean internal data on client side. You will get no information if the data is still valid or meanwhile has been changed. So you either should try to hold the data only for a short time to minimized the problem or you have to check for correctness of the data before you use it. Alternatively you can use some kind of time-stamp mechanism to check if the data is still correct. In some cases this is not necessary because either your clients don't need to always show the latest version of your data or the data held in your Value Objects is read only data and so can not be changed meanwhile.

Another problem with Value Objects is, that their number and structure is mostly driven through client use cases. So the interface of your bean depends heavily on client use cases and therefore might not be stable throughout your projects lifetime. To address this problem, look at the Value Object Creator and Hashtable Value Object patterns. Both patterns deal with the problem of decoupling the Component Interface from the need for new Valbue Objects.

Variants

A Value Object can have additional methods. These can be for example used to check consistency of the members. A special way to implement your Entity Beans is to move all members from the Entity Bean to a Value Object and use only this Value Object as member of your Entity Bean. Also your Entity Bean might inherit from a Value Object.

Value Objects can also act as a way to lazy-load additional Value Objects or to access the Entity Bean itself. For example, a Value Object containing only name information for a *Person* Entity could provide an operation *getEntity()*, returning the entity itself or it could provide operations to get additional Value Objects such as *getAdressInfo()* or *getJobInfo()*. Thus, different Value Objects can provide different views for an entity, while still providing access to the entity itself if this should become necessary. For example, a table view can use a Value Object containing the required information, and only when one of the displayed entities is selected for editing, only then is the entity accessed directly.

Related Patterns

The Bulk-Setter pattern addresses the same problems as Value Object but works only for setter operations. A Value Object is similar to a Dependent Object but has no semantical meaning.

Value Objects can be combined with a lot of other patterns out of this book. One example is the Bulk Reader pattern where the returned attributes need to be grouped when being returned to the client. Value Objects can be used for this.

Hashtable Value Object and Value Object Creator solve the problem that client use-cases have direct influence on the Entity Beans interface as Factory Methods for the Value Objects are needed.

Combined Value Object

A client needs a the state (or a part of the state) of several entities during one use case. In the usual approach, the client will contact each entity separately, querying the attributes (or preferably getting the respective VALUE OBJECTS). However, this involves several network hops and it requires client's knowledge about the relationships among the entities, and how to navigate them.

Consider an *Order* entity. An order consists of the data directly related to the order, and it has a 1:n association with several *Items,* whereas an *Item* has a reference to a *Product*. And, of course, an *Offer* has a reference to the *Customer* who issued the *Order*.

Two different views of such an *Order* seem useful. One can be used in a tabular view of several orders. You can use BULK READER to get a list of VALUE OBJECTS for *Orders*. The other is for looking into the details of one *Order*. Using the standard, VALUE OBJECT based approach, you would access the *Order* to get its VALUE OBJECT, then access the *Items* and their associated *Products*, and then the *Customer*, requesting VALUE OBJECTS from each of them in turn. Once again, this results in many network hops that should be reduced.

In addition, the client programmer needs to know the relationships among the entities and has to navigate them manually.

Therefore:

Provide a special Session Bean that collects data from several, related entities and creates a COMBINED VALUE OBJECT, which is then returned to the requesting client.

❊❊❊

Advantages: This pattern helps to build use-case- or client-specific VALUE OBJECTS for several entites, however, the different entities do not directly depend on each other. The *Facade* session bean knows about the relationships only.

The consequence of using this pattern is improved performance because of a reduced number network hops and it reduces the dependencies among the entities. Because these COMBINED VALUE OBJECTS are created "externally".

Drawbacks and Caveats: Of course, on the downside, the client programmers need to know, that there is an additional component that provides the required aggregated information. And, of course, the COMPONENT which creates the COMBINED VALUE OBJECTS depends on the entities it uses – but this is not a problem, because this SESSION COMPONENT is usually use-case specific anyway.

If you need to use this pattern extensively, you should review your architecture, because your COMPONENTS could be too fine-grained, and you should use DEPENDENT OBJECTS instead to organized the state.

Related Patterns: This pattern is especially useful if it is used in the context of a BUSINESS COMPONENT, which uses a *Facade* to shield its internal structure from its clients.

If the contents of these COMBINED VALUE OBJECTS changes regularly, use a VALUE OBJECT CREATOR for this session bean.

If a client requires many instances of a COMBINED VALUE OBJECT, consider using BULK READER.

Hashtable Value Object

You use VALUE OBJECTS to set and get subsets of the attributes of your Entity Beans. As you need to set or get many different permutations of attributes you also need many different VALUE OBJECTS and FACTORY methods within your COMPONENT. So you have many operations only dealing with VALUE OBJECTS.

Also if your client changes rapidly you often have to change the interface of your COMPONENT. This also leads to the need for frequent redeploys. So using many VALUE OBJECTS may lead to an unstable and bloated COMPONENT INTERFACE.

Imagine a Entity Bean that has many attributes for example a Entity Bean that represents a project in a distributed project management system. As you have different GUI dialogs that access subsets of the projects parameters you need to introduce different VALUE OBJECTS to access the attributes in a performant way. The Interface of your project Entity Bean gets loaded with methods to set and get the VALUE OBJECTS. This makes using your project Entity Bean complicated and error prone.

As an alternative you could implement a VALUE OBJECT holding all members of the Entity Bean. This is not a good solution either, because this would result in very large VALUE OBJECTS that have to be transferred to the client even if only a small subset is needed by the clients use case.

Therefore:

Insert the attributes as name/value pairs into a hashtable. Use the hashtable as a parameter to set or get the attributes in your Entity Bean. This is a so called HASHTABLE VALUE OBJECT. Using a HASHTABLE VALUE OBJECT you will get a small and stable COMPONENT INTERFACE to set a variable set of attributes.

❊❊❊

General

A HASHTABLE VALUE OBJECT is basically a simple form of a reflective data structure. A client can query the object for the attributes it supports. The number and type of the attributes is not predefined at compile time. As a consequence, this pattern is most useful if the client application can cope with these dynamically changing attributes. Typical applications of this kind are all those that interact with the user: A table view can easily add a column for each attribute discovered; a product information page can display all the attributes available for a specific product instance; an editor can be created dynamically, providing an edit widget for each discovered attribute.

Advantages

A HASHTABLE VALUE OBJECT brings the same performance advantages as a normal VALUE OBJECT, because many calls to an Entity Beans get or set operations are grouped together to one call.

Additionally you get a stable interface for your Entity Beans because a new set of attributes does not lead to the introduction of a new VALUE OBJECT. You don't have to change the COMPONENTS IMPLEMENTATION or to redeploy your COMPONENTS as long as the COMPONENT is able to deal with the name/value pairs from the caller. In addition, you don't need to write new VALUE OBJECT classes whenever the attribute sets that need to be get or set change.

This also has the advantage of less classes needed and therefore lower maintenance costs and smaller applications. This might be especially important if your Enterpriser JavaBeans client is an applet.

Drawbacks

A disadvantage of this pattern is, that you can not rely on type checking from the compiler anymore. You can implement type checking in your COMPONENTS operations, but then you shift type checking from compile-time to run-time. This leads to higher test and debugging efforts.

Variants

An implementation variation is that you don't use a hashtable but a class derived from hashtable to transfer the attributes. This gives you the possibility to add methods to the hashtable. This way for example you might check the arguments in your hashtable.

Related Patterns

HASHTABLE VALUE OBJECT is a variation of VALUE OBJECT.

Value Object Creator

The problem with VALUE OBJECTS is that client use-cases enforce the structure and number of VALUE OBJECTS needed. As described in the VALUE OBJECT pattern, you need a FACTORY operation for every VALUE OBJECT. Therefore you need to change the COMPONENT INTERFACE of your Entity Bean if your client changes its needs for members from your Entity Bean. This not only leads to frequent changes in the COMPONENT INTERFACE but you also need to redeploy your Entity Beans often. Also you have to update all other clients accessing the changed Entity Bean.
Having client use-case specific COMPONENT INTERFACES in Entity Beans, compromises their reusability. It also might result in interface bloat, and therefore should be avoided.

Imagine a Entity Bean representing a person. You created two VALUE OBJECTS to retrieve groups of members for the name (consisting of first name, middle initial and surname) and one for the address. For each VALUE OBJECT type you have a FACTORY METHOD in your Entity Bean creating them.

Now your client changes its GUI and offers a new dialog where users can edit a persons name and address simultaneously. So the client needs to retrieve data for the name and for the address for the new dialog in one step. You have to create a new VALUE OBJECT and a new FACTORY METHOD to get all the needed data within only one call. So you need to change the COMPONENT INTERFACE of your Entity Bean on behalf of client use-cases.

Therefore:

Add a generic FACTORY operation, a so called VALUE OBJECT CREATOR, to your Entity Bean. The VALUE OBJECT CREATOR takes an id as parameter which identifies the type of the VALUE OBJECT the client needs. Use the id to dynamically create the corresponding VALUE OBJECT.

Then call a method of the VALUE OBJECT itself that retrieves all necessary data from the Entity Bean. So this callback method lets the VALUE OBJECT fill itself with the data from the Entity Bean. At last step return the VALUE OBJECT to the caller.

***

General

Use some configuration mechanism such as CONFIGURATION PARAMETERS to map the id for the VALUE OBJECT to the class name of your VALUE OBJECT (see pattern: CONFIGURATION PARAMETERS).

Advantages

If you use VALUE OBJECT CREATOR you get a more stable COMPONENT INTERFACE at your Entity Beans because the interface does not dependent on client use cases that define which FACTORY operations for VALUE OBJECTS you need.

You can even add new VALUE OBJECTS at the runtime of your Entity Bean, as don't have to change the implementation of your Entity Bean. The only requirement is that the CONTAINER has to have access to the VALUE OBJECTS implementation.

Drawbacks

A disadvantage is that you introduce a slight performance overhead for retrieving the mapping information and dynamic creation of the VALUE OBJECTS. Usually this overhead is much less time consuming than doing many remote calls, if you wouldn't use VALUE OBJECTS at all.

You have less type checking at compilation time, because you need to downcast the VALUE OBJECTS at client side after using the VALUE OBJECT CREATOR. So your client should be able to deal with situations where through faulty deployment the server returns the wrong kind of VALUE OBJECTS.

Also VALUE OBJECT CREATORS implementation has a relatively complex structure. This makes your Entity Beans implementation harder to develop, maintain, and deploy.

Related Patterns

Some configuration mechanism, like described in CONFIGURATION PARAMETERS, is needed to configure the mapping between the id that describes the type of VALUE OBJECT and the class name of the VALUE OBJECT.

Another way to decouple the COMPONENT INTERFACE from client uses-cases is HASHTABLE VALUE OBJECT. HASHTABLE VALUE OBJECT is basically a reflective datastructre that can be adopted to the changing client needs. It is simpler in its implementation.

Bulk Reader

A client needs to display a set of entities in a tabular view, or wants to execute another kind of bulk retrieval, where only a small subset of the state of each entity is required. Usually, each entity (ENTITY BEAN or TYPE MANAGER) has to be accessed separately, creating a significant overhead in network traffic (and more, for ENTITY COMPONENTS).

A client application wants to display a list of all *Products* of a specific category. The displayed list should only show a *Product*'s ID, the name and its price. A naive approach for *Products* represented as ENTITY COMPONENTS would use the home interface and execute a finder operation, specifying the required category. The finder will return a list of references, each of them will be accessed in turn, requiring a logical COMPONENT instantiation in the CONTAINER and VALUE OBJECT retrieval for each of them. This takes up quite some performance and should be avoided.

For *Products* implemented with a TYPE MANAGER, the situation is somewhat better. The logical instantiation is avoided, but still, each entity must be accessed to retrieve VALUE OBJECTS in turn.

Therefore:

Provide a special SESSION COMPONENT that provides finder operations that access the database directly and return a list of VALUE OBJECTS for each of the found entities.

❊❊❊

Advantages: Using this pattern has the advantage that a query only retrieves the required part of the state of an entity that's really required, but for all required entites. This is done by directly querying the database, no entity has to be accessed. Performance increases significantly. Usually, the information returned is read-only, so there is no consistency problem.

Drawbacks and Caveats: **I**f the structure of the entity changes, you have to adapt not only the entity, but also the session which also accesses the (database) structures.

Variants: If VALUE OBJECTS are already used to access the state of the entities, these same VALUE OBJECT classes can be used to act as return values for the finder operations in the bulk reader. If only a subset of the values that are found by such a finder is actually needed by the client, you can use a SERVER SIDE ITERATOR to reduce network traffic, by transporting only those VALUE OBJECTS that are actually needed.

Additional Information: In the context of this pattern, the returned VALUE OBJECTS serve as a certain kind of descriptive proxy for the real entities. In many cases (consider again the tabular view of products) the complete entity will eventually be needed (e.g. to modify a specific order). In such a case, it is convenient for the client if the returned VALUE OBJECT provides an operation to return a reference to the complete entity.

Server Side Iterator

The result of a query is usually a collection of result objects (maybe references, or specific VALUEOBJECTS). If this collection is rather big, any the client may not need to access all entries, a lot of unnecessary data is transmitted from the server to the client.

An *OrderManagement* component (based on a TYPEMANAGER) provides an operation to retrieve all orders of the current week, sorted by descending date. The fitting entities are quite many, in the order of several thousands. The client application shows these orders to the user in a table view. The result of the query operation is a collection of VALUE OBJECTS (small ones, only containing the PRIMARY KEY and some descriptive information). In many cases, the user only need to see some of these result objects, usually the most recent ones. If the query operation returns all objects in one return, a lot of unnecessary data will be transmitted.

Therefore:

Create a Server Side Iterator in a SESSION COMPONENT. The client retrieves chunks of data from this iterator (i.e. a small collection of several result objects) at a time, requesting more data only if the data is really required. The server side iterator keeps track of what has already been requested by the client.

❊❊❊

Advantages: This pattern allows a client to access only the part of the data that is really necessary, avoiding unnecessary network overhead (and locking on the server), while still not requesting each VALUE OBJECT separately.

Drawbacks and Caveats: However, there are a couple of things that must be considered when implementing this pattern:

- For each such large query, a stateful session bean has to be created on the server. This bean either stores the complete collection and returns it step by step, or it uses techniques such as database cursors to delegate the "iteration state" to the database.
- The client needs to retrieve chunks of data from time to time, whenever it runs "out of data". If you do not want to make this explicitly visilble to your client, need a helper object on the client side, which, when requested for the next result object, automatically retrieves a new chunk, if necessary. A *Factory* on the client side can help to hide this.
- The session bean has to be removed sometime. A client must therefore explicitly call a "I am finished with this iterator" operation on the server side iterator, if it is not needed anymore. SESSION COMPONENT timeouts can help to clear stale iterators.

Depending on the way how the pattern is implemented, for example, when the queried values are cached in the SESSION COMPONENT because database cursors are not available, resource requirements for the server can be relatively high. In addition, changes in the database that have been committed after the query had been executed will not be reflected in the query results.

Variants: As a variant of this pattern, a SERVICE COMPONENT can be used instead of the SESSION COMPONENT if the client keeps track of the iteration state itself and the underlying database supports cursors. This results in a more lightweight solution for the server.

Entity Bulk Modifications

Updating a large set of entities in a similar way would need to access the respective Entity Beans in sequence. However, accessing Entity Beans is expensive because it is a two step operation: First you need to retrieve the primary keys and then in a second step you retrieve and then instantiate the Entity Beans. This often will be done with two database accesses.[3] Additionally every call to an Entity Bean will be intercepted by the container to check security and transactional settings. As all this will be done for every single Entity Bean, performance will degrade.

You implement an Enterprise Resource Planning System. Your *Employee* entities are implemented as Entity Beans. Now you need an operation to increase the salary of all employees in your company. You add a method *increaseWages(percentage)* to your *Employee* entity bean. Then you implement a Session Bean that calls this operation on every employee Entity Bean. This operation takes prohibitively long to complete.

Therefore:

Use a Session Bean that has an operation that modifies the entities directly in the database. Do not access the Entity Beans representing the entities at all. The updates are executed directly on the database, the Entity Beans are not involved.

❊❊❊

ENTITY BULK MODIFICATION is not a replacement for Entity Beans, but a way to deal with large modifications on bulk data. In contrast TYPE MANAGER is a complete replacement for Entity Beans.

If you use ENTITY BULK MODIFICATIONS you create a much lower load on your application server and your database. This is because your application server does not instantiate the single Entity Beans anymore. The databases load will be reduced because you do not read the data (maybe even twice) and then update it, but directly update the data in the database given some criteria. The database can use its query optimizer to improve performance.

Nevertheless, your ENTITY BULK MODIFICATION may take a long time to finish. A reason for this might be that some data that is represented by your Entity Beans might be locked in your database, because some of the Entity Beans holding data from the table you want to modify are actually used at the time, you start your ENTITY BULK MODIFICATION. Your ENTITY BULK MODIFICATION operation may have to wait until the locks are released.

As your ENTITY BULK MODIFICATION also holds a lock on the table or some rows of your database while it is running (depending on the locking technique your database uses), you might get serious performance problems and maybe even time-outs accessing Entity Beans stored in this table. Take care on these side effects of ENTITY BULK MODIFICATION when you need to modify many entities at a time.

Another problem with ENTITY BULK MODIFICATION can occur if you use some kind of caching in your setup. If your cache does not recognize the direct change of data in your database your Entity Bean might work on old data until your cache does the next synchronization with your database. Remember that Entity Beans itself are "cached" instances of the entities from the database.

Also, as you have to add database code to your session bean, your code can not be database independent anymore even if you use container managed persistence, so you sacrifice portability for performance. Especially when you use Container Managed Persistence you need to know how the container actually maps

3 Actually when you use Container Managed Persistence your Container could reduce this to only one database call. Nevertheless most CONTAINERS do not implement this optimization right now and do an extra database access for the retrieval of the PRIMARY KEY and the retrieval of the actual data of the bean. Also if you use Bean Managed Persistence your Entity Bean access will be mapped to two database accesses as described above.

the data of your Entity Beans to the database.

The EJB 2.0 specification introduces so-called *home business methods*. These methods are executed on a bean in pooled state and therefore cannot access state from a specific bean. This is the EJB equivalent to static methods in Java. If you use EJB 2.0 you should implement your Entity Bulk Modification operation as a home business method. This should be done, because Entity Bulk Modifications are conceptually a part of the Entity Bean.

Entity Bulk Modification is a pattern for updating big chunks of data in a similar way. If you need to read a large number of entities use Bulk Reader. If your client application needs to traverse through a large number of entities use Server Side Iterator.

Patterns for Component Variability

Quick Access Table

if...	use...
... a new component should offer several interfaces	EXTENSION INTERFACE: The actual interface of the component gives access to serveral interfaces
... an existing component should offer several interface	MULTIPLE INTERFACES: The implementation of the component is deployed with several different interfaces.
... the interface of a component changes frequently	WEAKLY TYPED INTERFACE: Provide just one generic method in the interface that can handle the arbitrary requests.
... the behavior of a component should be changeable in certain parts without modification of the component itself	BEHAVIOURAL FLEXIBILITY (aka STRATEGY): Delegate the parts of the functionality to another class that is loaded by Java's dynamic class loading.
... additional attributes should be stored in the component without modification of the component itself	ATTRIBUTE LIST: Provide generic methods to set and get arbitrary attributes.

Attribute List

Your components feature a set of attributes which are accessible through hard-coded accessor operations, such as setter/getter pairs or VALUE OBJECTS. However, depending on the use of the COMPONENT, additional data has to be stored with the COMPONENT, i.e. additional attributes are necessary. You do not want to change the COMPONENTS implementation every time a new attribute is required, perhaps only for a specific use case.

In a hospital information system, a *Patient* entity has a set of mandatory attributes, such as name, date of birth, blood group, etc. In addition to these attributes, each hospital (i.e. installation of the system) has additional requirements. For example, they want to store the date when the patient has been in the hospital for the last time, or the family doctor. The patient entity must be able to store these additional, freely defineable attributes.

This is especially important in situations where such attributes are stored to provide logical connections to external systems.

Therefore:

Provide a COMPONENT with an attributes list, a set of name-value pairs, which can be accessed by *setAttribute(name, value)* and *getAttribute(name)* operations (or their bulk-accessor optimizations).

❊❊❊

Advantages: This pattern allows you to store any kind of additional information with a COMPONENT. This allows developers to customize the entity depending on the use case.

Drawbacks and Caveats: The disadvantage of this pattern is, that you have two different programming models for attribute access. Some attributes, the regular ones, are accessed using hard-coded operations like *getID()*, *getName()*, etc. while the attributes in the ATTRIBUTE LIST are accessed using the *getAttributeValue()* operation. Also wrong attribute names can not be checked by the compiler i.e. access to a non-existing attribute is not detected until the code is actually executed. Even if the attibute exists, further errors can occur. As they are returned as generic types the type must be casted. These casts are another source for errors. So an ATTRIBUTE LIST is more error prone than the normal attributes. This might cause additional testing effort.

Variants: In the simplest case, these *set/getAttribute* operations allow any attribute to be set or read. In a more advanced version, the list of allowed (or even mandatory) attributes can be defined using CONFIGUATION PARAMETERS. The get and set methods must return generic types such as *String* or *Object*.

Another situation where you can use this pattern is if your COMPONENTS have a large set of attributes that could potentially be used, but only a small fraction of these attributes is used by each instance.

Additional Information: Because entities are persistent, you must make sure that these attributes can be stored as part of the component's state. In an OODBMS this is no problem, because the attributes are stored in the list directly. In relational databases, you either have to implement a vertical data model (using a table with name-value pairs) or you can store the values as a BLOB (e.g. in the form of *name1=value1;name2=value2;...).* The latter has the drawback that you cannot easily search through them.

Extension Interface

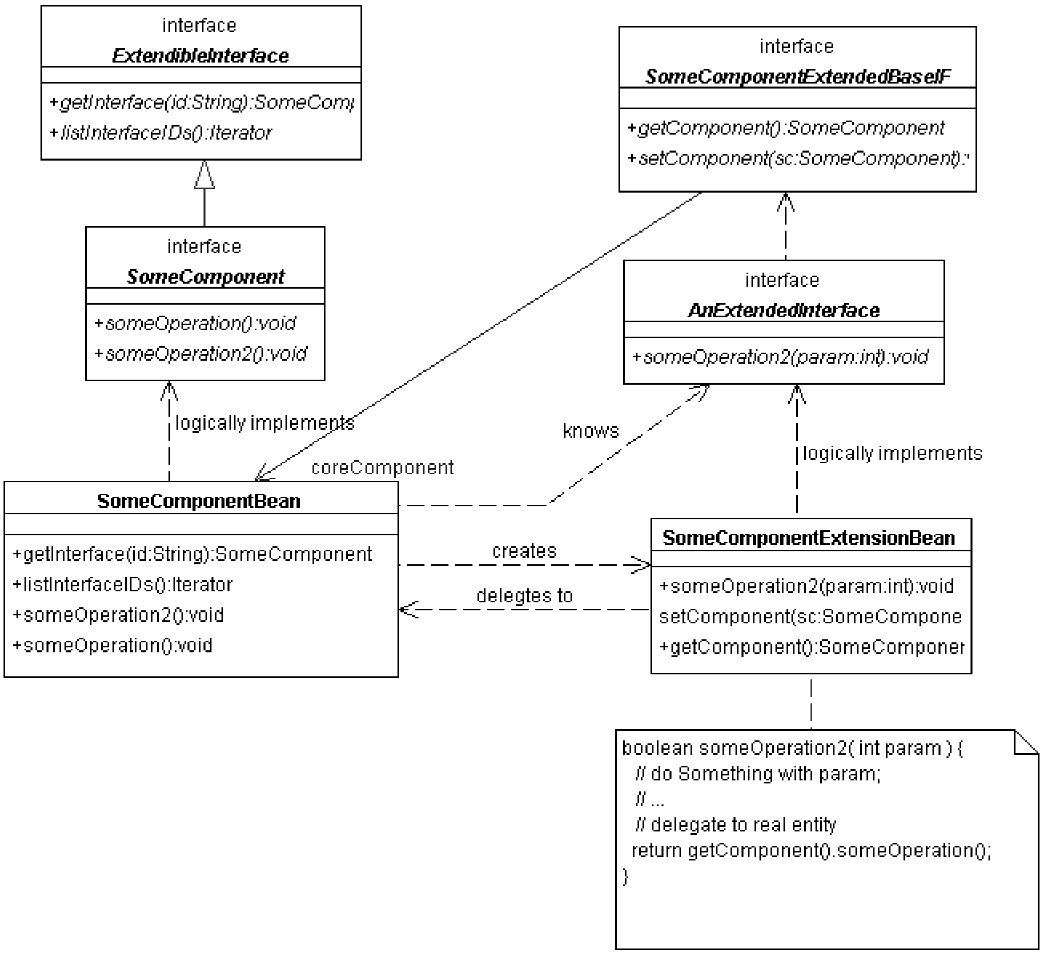
A COMPONENT should usually be responsible for a well-defined task. In some situations, however, this task cannot be specified by exactly one interface, because different clients have different "views" of the task, or the COMPONENT simply plays different roles (requiring different interfaces). A common problem is that interfaces change over time, and you want to keep old interfaces available while still allowing new interfaces to be added.

Imagine an application, where you need to evolve interfaces over time, by adding or removing business methods, e.g. because they are not supported anymore. Old clients should still be able to access the old interface as before, but new clients should access a new interface, on the same COMPONENT. You cannot use interface inheritance, because you want to change or remove behavior from the COMPONENT. COMPONENT inheritance is forbidden, and conceptually questionable.

This approach is especially important for ENTITY COMPONENTS, which, by definition, provide a representation of a business entity. The actions, or processes, possible with such an entity usually vary significantly more than their data content.

Therefore:

Separate the operations interface for use by the clients from the COMPONENT itself. Let the COMPONENT manage the interfaces it supports explicitly, and allow the clients to query the COMPONENT for a specific interface based on a key (perhaps incl. version tag). The clients can then use this interface to access the COMPONENT.


interface
ExtendibleInterface
+getInterface(id:String):SomeCom
+listInterfaceIDs():Iterator
interface
SomeComponentExtendedBaseIF
+getComponent():SomeComponent
+setComponent(sc:SomeComponent):
interface
SomeComponent
+someOperation():void
+someOperation2():void
interface
AnExtendedInterface
+someOperation2(param:int):void
logically implements
knows
coreComponent
logically implements
SomeComponentBean
+getInterface(id:String):SomeComponent
+listInterfaceIDs():Iterator
+someOperation2():void
+someOperation():void
creates
delegtes to
SomeComponentExtensionBean
+someOperation2(param:int):void
setComponent(sc:SomeCompone
+getComponent():SomeComponer

```
boolean someOperation2( int param ) {
  // do Something with param;
  // ...
  // delegate to real entity
  return getComponent().someOperation();
}
```

⁂

Additional Information: The extensions are themselves Session Beans (using Service Beans is not possible, because they must keep track of their core Component) and they delegate their operations to the core Component eventually. If new operations on the core Component are really necessary (e.g. because more state is necessary for Entity Components) then this need to be reflected in the old extensions. Thus, old applications cannot access this state.

Usually you will not give clients access to the core Component. The pattern is most useful if all client interaction is achieved through extensions.

Advantages: This pattern allows the interface(s) to evolve independently from the Component implementation itself. It is even possible to add new interfaces to a Component without requiring a new reinstallation. The Component must know the set of available extensions, for example by using Configuration Parameters. The mechanism to query the Component for its interfaces, and the creation of the necessary extension beans can be implemented generically, as the example below shows.

This pattern is a powerful way to prevent interface bloat, especially for deprecated operations. The pattern also allows for a role-based development, because several completely unrelated Components can have the same interfaces. It is important to see, that these extension interfaces can be used for permission control.

Drawbacks and Caveats: On the downside this pattern requires "returning" extensions that are not required anymore – or Session Component timeouts will fire.

Related Patterns: Basically, this pattern is a kind of *Adapter* [GHJV94] (actually, an object adapter) with the extension, that the adaptee manages the available adapters. An alternative way to give several interface to one Component is to use MULTIPLE INTERFACES.

This pattern is described in depth in [POSA2]. It can also be seen as an application of the Role Object pattern [DR??] which is in turn a kind of Adapter [GHJV94].

Multiple Interfaces

Every time an interface changes all parts of the system that use the COMPONENT must be changed as well, at the very least they must be recompiled and therefore also redeployed. This means a small change might cause a cascade of changes.

Imagine that a new method should be added to the Remote Interface of a Session Bean. Despite that the new version could still support the old Remote Interface, a new Remote Interface must be written. This in turn causes all old clients to fail when they try to access the COMPONENT. So they must be recompiled with the new Remote Interface. In a high availability scenario this is not acceptable at all but even in "normal" applications a lot of clients might need to be recompiled and this can lead to unnecessary delays.

Therefore:

Instead of changing the interfaces provide a new (changed) interface and at the same time keep the old one available as well. Both interfaces share the same implementation class and also access the same data in case of an Entity Bean.

❊❊❊

General

Most Component architecture directly support that a COMPONENT might have multiple COMPONENT INTERFACES because different clients might need different interfaces. Usually an old interface must be still be offered while the new clients already use the new interface. Enterprise Java Beans has no direct support for such a feature. However, there is no technical reason for the implementation of a Session or Entity Bean to offer just one interface. An implementation can be created that supports multiple interfaces and thus allows easy evolution of the interfaces. It is a good idea to add a version tag to the interface name and the JNDI name for the Bean.

Of course this doesn't solve the problem of offering different version of the implementation. I.e. if the implementation of a method should be changed it is not enough to have the same implementation offer two different interfaces. However, the implementation can retrieve the Remote Interface it was called from and thus offer version specific behavior.

Related Patterns

Compared to EXTENSION INTERFACE this pattern allows a second interface to the same component while EXTENSION INTERFACE creates additional components that have a reference to the original component and thus can offer additional functionality.

Multiple Interfaces can be used to implement ADMINISTRATABLE COMPONENT.

Behavioral Flexibility (aka Strategy)

You want to keep some aspects of your COMPONENT'S behavior flexible to facilitate reuse. This flexibility does not influence the interface of the COMPONENT. The flexibility should be configurable at COMPONENT INSTALLATION and new behaviors, which the implementer did not foresee should be possible.

As an example, imagine a COMPONENT which has to notify other components/users when a certain state occurs. This notification can be achieved on different ways:

- another component can be called
- a JMS message can be sent
- an email can be sent
- a pager can be used to notify a user
- or other, yet unknown means should be allowed.

You do not want to leave it to the user of the COMPONENT to choose a suitable behavior, and you don't want to change the implementation of the COMPONENT if a new kind of behavior is required.

This is especially important for COMPONENTS that should be sold as a product, but where certain aspects of the behavior should be kept flexible.

Therefore:

Use the Strategy pattern [GHJV94] together with Java's dynamic class loading to realize different behaviors inside of a bean. Use CONFIGURATION PARAMETERS to specify the actually required class name. If only a certain set of behavioral options should be allowed, use keys to specify them in the CONFIGURATION PARAMETERS.

❊❊❊

Advantages: This pattern allows you to "introduce" any kind of new behavior into a component, in places, where the original component developer planned for it. However, please note that "wrong" behavior can compromise the integrity of the host component.

Variants: There are basically two ways how concrete classes can be supplied to the COMPONENT. One is to use CONFIGURATION PARAMETERS to specify the class name, and the COMPONENT uses dynamic class loading to create an object. Alternatively, the client can pass an object to the COMPONENT, which requires that the corresponding class is also uploaded to the COMPONENT.

Additional Information: This might be a basic technique for a component marketplace because it allows the component customer to plug in his own code into an existing component. This enables more powerful customization of components and thus allows a broader variety of applications for components.
This is basically an adaptation of GoF's strategy pattern [GHJV94] for the EJB world.

Weakly Typed Interface

Adding new functionality to a COMPONENT requires changes in the COMPONENT INTERFACE. This usually also requires recompilation of the clients in order use the new features. Redeployment is also necessary, because the contents of the CLIENT LIBRARY will change (new interface classes, etc.). This is unacceptable in many applications, especially high-availability systems.

Imagine a client application where the user can select certain calculations to be executed on the sever inside a *CalculationComponent,* perhaps as part of an Application Service Providing solution. It is easy for the ASP provider to update the implementation to provide new services. But changing the interfaces (which are distributed to each client!) is more difficult.

Thus, the ASP provider would prefer an interface which is stable over time, while still allowing new features to be added and used by the client. The latter requires that a client application can dynamically find out, which operations are supported by the COMPONENT.

Therefore:

Create a generic interface which has an operation that allows to generically specify commands, including

parameters. The component implementation can then interpret this command and return the results accordingly. To make sure the client can really work with new operations, reflective features must be added to allow the client to query the component for available commands.

❊❊❊

Advantages: Using this pattern allows you to provide new functionality without modifying the interface of a component, although the implementation has to be modified, of course. Redeployment is usually also necessary.

Drawbacks and Caveats: The pattern has some additional drawbacks. Using normal interfaces, the compiler can do type checking – you can only execute the operations that are allowed, with the correct set of parameters. Using weakly-typed interfaces will short-circuit these checks. So the pattern can lead to dependencies that are not visible in the interfaces. If a generic, reflective client is used, these problems will not appear.

Variants: A weakly typed interface which uses some ubiquitous format such as XML (possibly plus HTTP) can also serve as a very simple bridge to a subsystem implemented in another COMPONENT technology, such as EJB to COM+.

Additional Information: The pattern should only be used, if the client is generic in nature, i.e. if it can work with newly discovered commands, and integrate them into its functionality. GUI based applications seem especially suitable. To make the clients able to use the feature, more effort might be necessary than is obvious.

Some infrastructures for distributed systems have an implementation of this pattern built-in. For example, CORBA has the Dynamic Invocation Interface (DII) to dynamically create calls to arbitrary methods. Together with CORBA's Interface Repository this allows completely dynamic calls to any CORBA object without knowing the interface beforehand. CORBA also offers the possibility to dynamically handle remote calls by using the Dynamic Skeleton Interface (DSI). This allows a remote object to dynamically implement an arbitrary remote interface.

Related Patterns: A weakly typed interface is also useful as an interface for BUSINESS COMPONENTS, because is a suitably general request description is used (such as XML) integration is significantly simplified.

Acknowledgements

Writing that many patterns is quite a lot of work, and several people helped significantly. First of all, we'd like to thank our PLoP shepherd Ali Arsanjani, who provided us with many useful comments – unfortunately we could not integrate all of them because of time constraints. They will go in the next version, however.

In addition, we'd like to thank our colleagues at MATHEMA AG, who inspired some of the patterns and helped to review the drafts.

Literature and Online Resources

[AG00]	Alan Gordon, *The COM and COM+ Programming Primer*; Prentice-Hall, 2000
[AOP]	*The Aspect Oriented Programming Homepage*, http://www.parc.xerox.com/csl/projects/aop/
[BOF]	Herzum, Sim; *Business Object Factory*; Wiley
[CETUS]	Schneider, et. al., *Cetus-Links*, http://www.cetus-links.org
[DR00]	Bäumer, Riehle, Siberski, Wulf. "Role Object." In Pattern Languages of Program Design 4. Edited by Neil Harrison, Brian Foote, and Hans Rohnert; Addison-Wesley, 2000.
[GHJV94]	Gamma, Helm, Johnson, Vlissides; *Design Patterns*; Addison-Wesley 1994
[HDSC]	IBM Corp, *Hyperspaces*, http://www.research.ibm.com/hyperspace/
[ISE]	ISE, *An introduction to Design by Contract*, http://www.eiffel.com/doc/manuals/technology/contract/index.html
[JiniPL]	*The Jini Pattern Language*, http://www.cs.wustl.edu/~mk1/AdHocNetworking/
[JS00]	Jon Siegel, *CORBA 3*, Wiley, 2000
[JSP00]	Subrahmanyam, et. al, *Professional Java Server Programming* ; Wrox Press, 2000
[KJ00]	Michael Kircher, and Prashant Jain, *Lookup Pattern*, EuroPLoP 2000 conference, Irsee, Germany
[MH00]	Richard Monson-Haefel, *Enterprise Java Beans*, Second Edition; Wiley, 2000
[OMGCCM]	OMG, *The CCM Specification*, available from http://www.omg.org
[OMGREL]	OMG, *Relationship Service Specification*, available from http://www.omg.org
[POSA]	Buschmann, Meunier, Rohnert, Sommerlad, Stal; *Pattern-Oriented Software Architecture*; Wiley, 1996
[POSA2]	Schmidt, Stal, Rohnert, Buschmann; *Pattern-Oriented Software Architecture, Volume 2*; Wiley, 2000
[PS99]	Peter Sommerlad, *Configurability*, EuroPLoP 99 conference, Irsee, Germany
[SUNEJB]	Sun Microsystems, *EJB Specification*, available from http://java.sun.com/products/ejb
[Wo00]	Eberhard Wolff, *EJB und das Java-Typsystem*, Java Spektrum 06/00