New Life for EJB

(Updates since version 1.2)

Rajat Taneja and Ganesh Prasad

20 August 2004

The material in this document supercedes some sections of the document "New Life for EJB" version 1.2, and should be read together with that document. A new version of the complete document incorporating all these updates will be released soon.

Table of Contents

The EJB Framework – Basic Design Concerns 3
- Introduction 3
- Architectural coupling and dependencies 3
- The remote/local issue that won't go away 8
- Performance and security issues 10

The Solution 12
- Solution description 12
- Don't Transfer Objects violate encapsulation? 18
- What about performance? 18
- What about co-located “remote” clients? 18
- What about ease-of-use and testability? 19

Conclusion 20

The EJB Framework – Basic Design Concerns

Introduction

We thank readers of our document "New Life for EJB v 1.2" for their many questions, suggestions and criticism. We responded to some comments directly, but many others required more thought and rework. Most of the negative comment related to our model for Data Transfer Objects (DTOs). We have accordingly overhauled this aspect. There are other changes as well, relating to ease of use and testability outside the container. We think these changes should make our model much more attractive.

The primary thrust of the changes to our model is the issue of access to data, and this will be the primary focus of this document. Designing a sensible architecture for EJB requires attention to several orthogonal aspects of data use:

- Architectural coupling and dependencies
- Remote versus local issues (Pass-by-value versus pass-by-reference semantics)
- Performance Issues – Serialisation, Copying and Network Bandwidth
- Security Issues – Confidentiality/Privacy
- Ease of use and testability

Architectural coupling and dependencies

EJB (Enterprise JavaBeans) means many things to many people, but to architects, EJB represents one of the most modern ways to model and build a complex, distributed system containing data, process and event components. The promise of EJB (so far not optimally realised) is to provide an elegant and efficient way to translate well-designed distributed systems into working applications.

To understand EJB, one must look at it not so much as a set of concrete components and technologies, but as a means of primarily *modelling* enterprise systems and secondarily *translating* that model into a working application. Figures 1 and 2 illustrate this point.

Figure 1 – EJB-based modelling (ideal), showing dependency relationships

The data-specific Business Interfaces together describe the Domain Data Model. The Service-specific Business Interfaces and the Data View Interfaces (i.e., compound data elements exposed by those Services as parameters or returned values) together make up the Service Contract with external clients. Note that Data View Interfaces are not part of the Domain Model but the Service Contract.

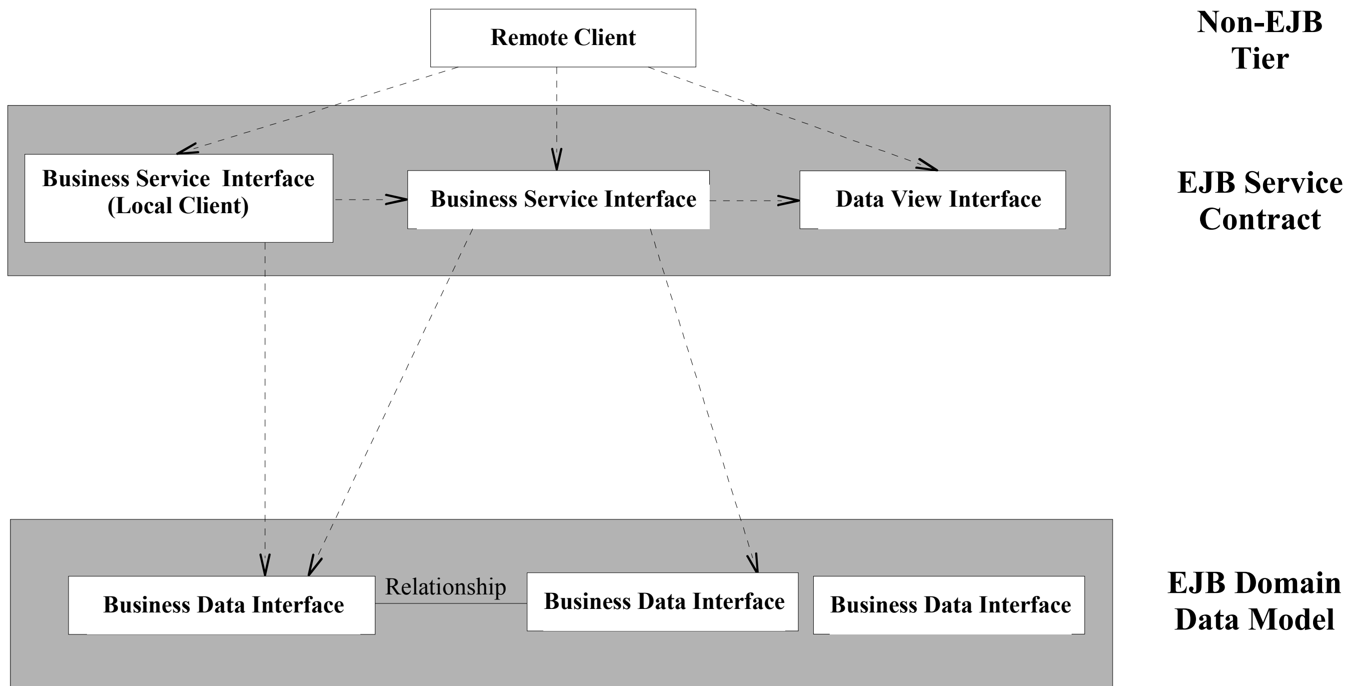
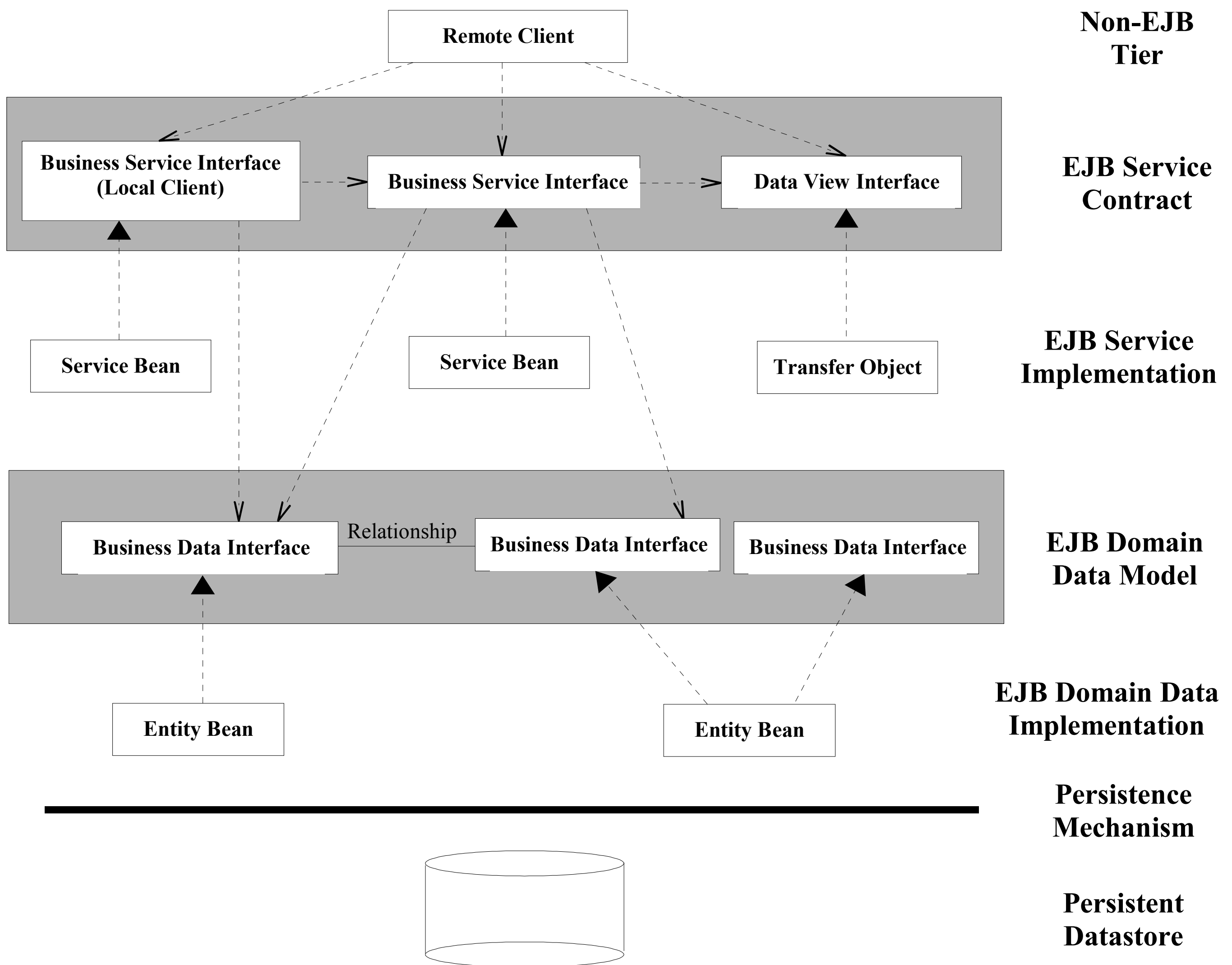


Figure 2 – EJB-based implementation (ideal), showing specialisation relationships

Concrete EJB components (beans) are used to implement the data and service aspects of the model shown earlier. Note that Transfer Objects are not part of the Domain Data implementation but the Service implementation.



All too often, IT professionals get caught up in the implementation aspects of EJB technology and fail to appreciate its domain modelling capabilities. The real contribution of EJB technology is in its dual ability to model and then to implement large, complex, distributed systems.

Modelling: Figure 1 shows the modelling artifacts of EJB, which are nothing more than plain Java interfaces. We call these "Business Interfaces", because they model the business domain. It is a well-known principle of OO design that domains must be modelled through interfaces and not through concrete classes. Concrete classes are clumsy artifacts from a modelling perspective. They must only be used to *implement* what the interfaces model. Interfaces represent explicit contracts between components. Concrete classes honour these contracts in a guaranteed way.

The EJB approach involves modelling two interface layers – a Domain Data Model and a Service Contract. The Domain Data Model (or Domain Model for short) describes the data organisation of the application – the business attributes in the system, how they are grouped, and how those groups relate to each other (cardinality). The Service Contract exposes the behaviour of the system as a whole to clients without revealing its internal design. This encapsulation is a good thing for clients, because the Service Contract shields them from changes to the internal design (where "internal design" refers not only to how Services are implemented but also what the Domain Model looks like and how *that* is implemented). In implementation terms, the Service Contract corresponds to the Session Façade pattern.

In a decoupled, service-oriented universe (which is where modern enterprises should be), the Service Contract is the only thing that a client should know or care about. The Domain Model is abstracted by the Service Contract and is freely modifiable without affecting the client, provided the Service Contract is honoured (in syntax as well as semantics).

Note that the application makes no guarantee to the client that the Domain Model will be stable. Visibility implies dependency, so the application-client contract only guarantees a stable Services interface. If a client can see elements of the Domain Model, it can then create dependencies that will break if the Domain Model changes without notice.

Implementation: Figure 2 shows the implementation artifacts of EJB technology layered onto the earlier modelling diagram. It can be seen that the presence of Entity Beans and Service (Session) Beans are really secondary to the basic model. They merely implement the Domain Model and the Service Contract, respectively. The power of the system is in the modelling. One of the nice things about the Entity Bean implementation of the Domain Model is that it abstracts out persistence concerns from the rest of the system. The Service tier is completely shielded from any knowledge of persistence (i.e., how it is managed and implemented). This is an EJB implementation feature that aids decoupled design, because Services can be assumed not to have any dependency on persistence.

Further elements such as event components (not shown in this diagram) may be layered on top of the Service tier to model the application's response to asynchronous events like messages and timeouts of various kinds.

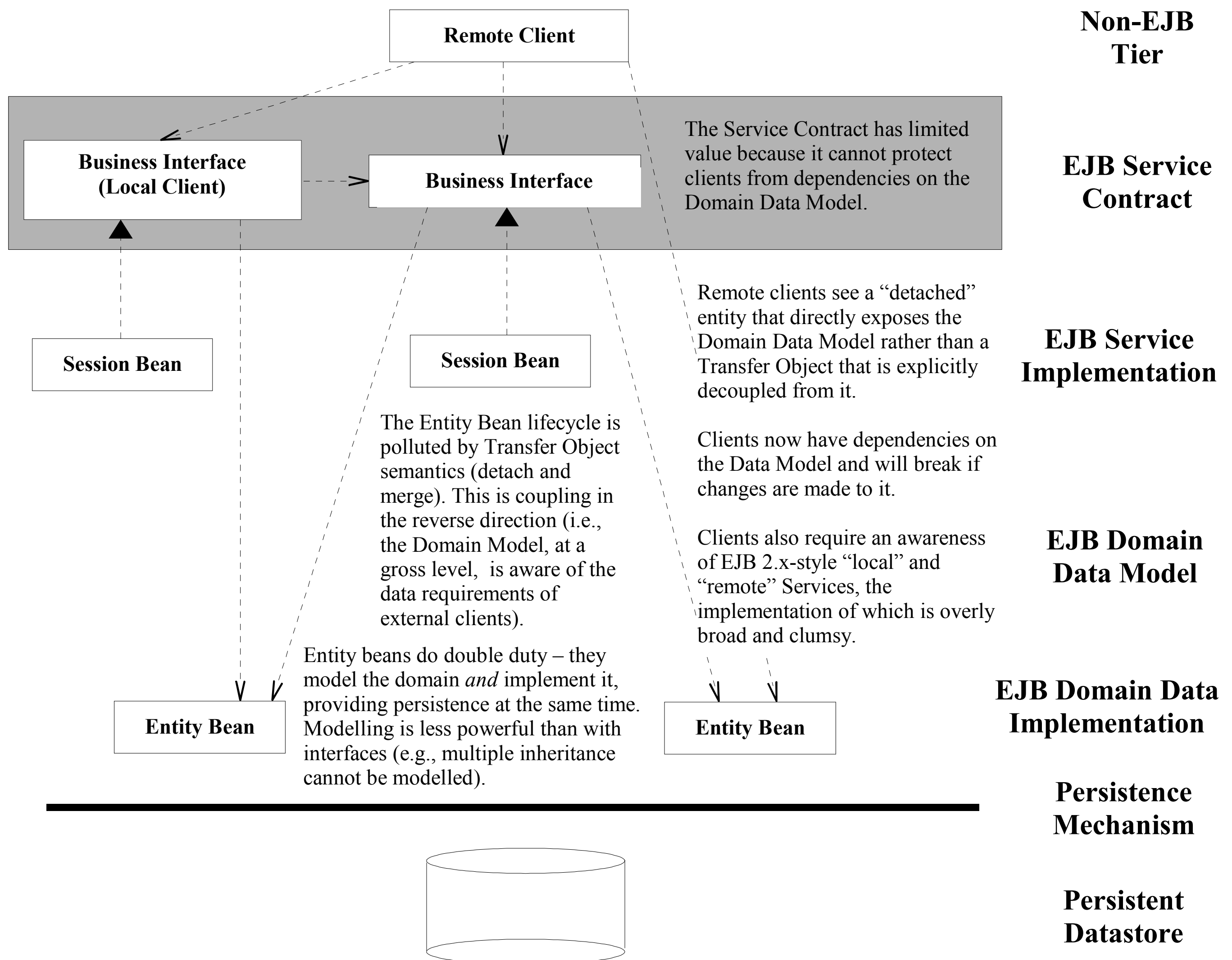
The EJB approach thus supports an extremely clean separation of concerns, but it is not idiot-proof. It is important that designers exploit its architecture rather than violate it through poor design or implementation.

In the past (EJB 2.x and earlier), the EJB framework itself was not well-architected, although in concept, it allowed application developers to model their own applications well. This meant that application architects could model their applications quite powerfully using Java interfaces, but when they turned to EJB technology to implement that model, it turned out to be clumsy in many places.

We believe that if the fundamental architectural problems with EJB technology itself are resolved, it can deliver on its promise to "provide an elegant and efficient way to translate well-designed distributed systems into working applications".

Unfortunately, the proposed EJB 3.0 specification has failed to do this. In fact, it seems to have placed "developer-friendliness" so far above correcting EJB's architectural problems that it ends up cutting many corners in the process. The resulting loss of modelling power and the tight coupling that it encourages render it far less capable as a framework for enterprise-class distributed systems. Figure 3 shows why the EJB 3.0 proposal fails to measure up to the enterprise yardstick.

Figure 3 – EJB 3.0-based modelling and implementation (As proposed by the early draft spec)



It is clear that EJB 3.0 has sacrificed too much to keep its programming artifacts simple. We show that it is possible to make EJB simple without such sacrifices. It is possible to build large, complex, distributed applications by following sound software and OO design principles, while keeping complexity to an absolute minimum.

It may be argued that it is still possible for application designers to use interfaces to model their Entities under the EJB 3.0 model. However, the detached entity mechanism does not exploit these interfaces, so the practical benefits of this are limited. (For that matter, it is possible for a developer to avoid interfaces when using our model, but we would strongly recommend against such an approach!)

The remote/local issue that won't go away

One of the fundamental problems with distributed systems is that they can never be totally location-transparent. This is not just an issue of network latency or bandwidth impacting performance. When object references are passed between components, the receiving component either gets a reference to the original object or to a serialised copy, depending on whether it is local or remote with respect to the sending component. This is because object references only make sense within the VM that houses the object, so servers generally serialise objects and pass them “by value” to remote components. The receiving component itself doesn't know about the switch.

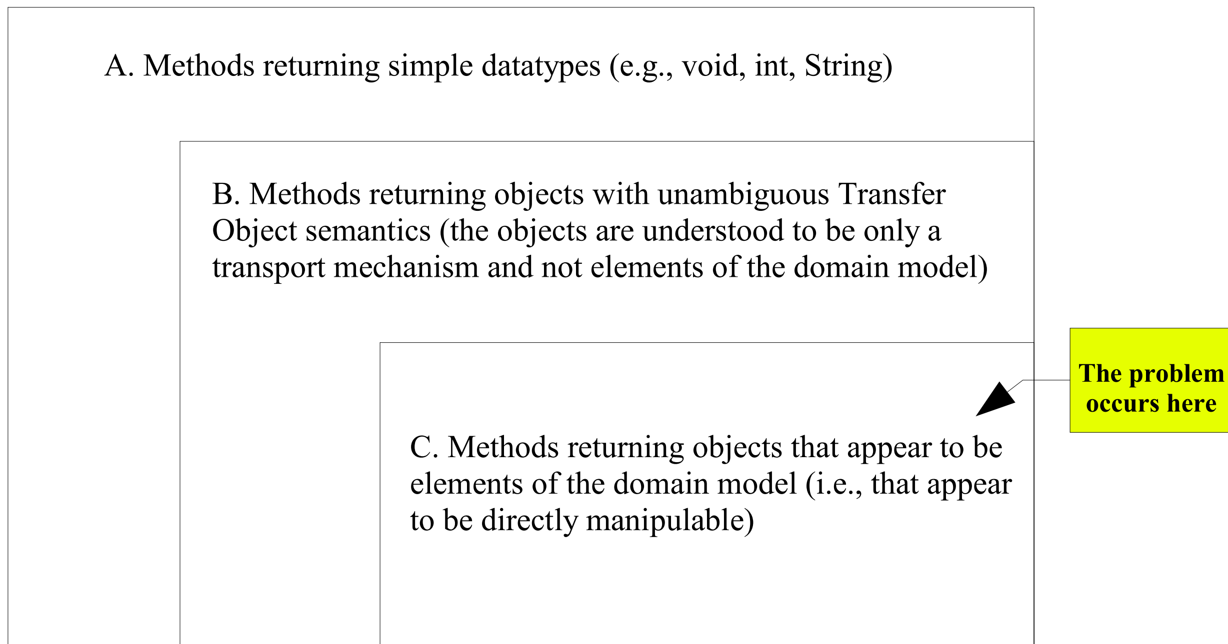
The practical upshot of this behaviour of distributed systems is that receiving components cannot assume that they have a reference to the original object (or for that matter, that they have a reference to a copy). If they proceed with one of these assumptions and it turns out to be false, the system as a whole will exhibit strange behaviour that is hard to debug, although it may not throw any obvious exceptions. We need a fail-safe mechanism to protect applications from such behaviour.

EJB 2.x solved this problem with what we believe was an overly blunt instrument. It defined what it called “local interfaces” and “remote interfaces” to Session beans themselves. Any reference that a Session Bean method returned through a local interface could be counted on to be a reference to the original object itself. Any reference that a method returned through a remote interface could similarly be guaranteed to be a reference to a copy. That provided a safe way for applications to tell whether they were dealing with originals or copies, and made applications much less likely to exhibit strange behaviour when client components were moved to different tiers (local to remote or vice-versa). However, it forced developers of client applications to deal with an unnatural and ugly programming construct that had nothing to do with business logic.

EJB 3.0 follows exactly the same approach, albeit with a different syntax. It uses annotations within the client component to specify whether a local or remote Session Bean interface must be used.

We propose a more elegant solution to this problem. Figure 4 analyses the different types of methods that a client component may call on a Session Bean, and the types of objects that may be returned. We can categorise them into three groups, and only one of these is susceptible to the kind of confusion and consequent dangerous behaviour that we described. The solution needs to be applied locally to these methods alone, not globally to all methods.

Figure 4 – Where the local/remote problem really bites



As can be readily verified, methods that return simple datatypes will behave the same regardless of whether the client is located locally or remotely. Even methods that return more complex objects are not automatically susceptible to behavioural changes. The *semantics* that the application applies to the object are of paramount importance. If the returned object is understood to be a “Transfer Object”, then it really doesn't matter whether the reference is to the original object or to a copy, because the client component will never try to update a Transfer Object as an end in itself. The Transfer Object, even if updated, will always be passed back through another method to update the Domain Model. The semantics of Transfer Object usage are so clear and well-understood that there is no danger of the pass-by-value/pass-by-reference issue making a difference to the way the application behaves.

That leaves the third group of methods, which return (or seem to return) elements of the Domain Model itself. Such elements invite clients to update them directly, because they imply such semantics.

We need to tackle the local/remote issue only for this group of methods. Imposing local and remote interfaces on beans themselves is overkill. We will show later on, when we describe our solution, how we tackle this problem without requiring the use of local and remote interfaces.

Performance and security issues

A major performance drain on distributed components comes from the serialisation of objects when they are transported across a network. This insight has led to a needless argument between proponents of architectural purity and those of efficiency and performance (It is needless because our model reconciles both concerns). Figure 5 shows the naïve view of what "local" and "remote" mean, and Figure 6 shows what they correspond to in the real world.

Figure 5 – The Naïve View of Distributed Components

The naïve view of distributed components is that they are either "local" or "remote" with respect to a given component, depending on whether they are on the same tier or a different tier. The problem is that the term "tier" has different meanings.

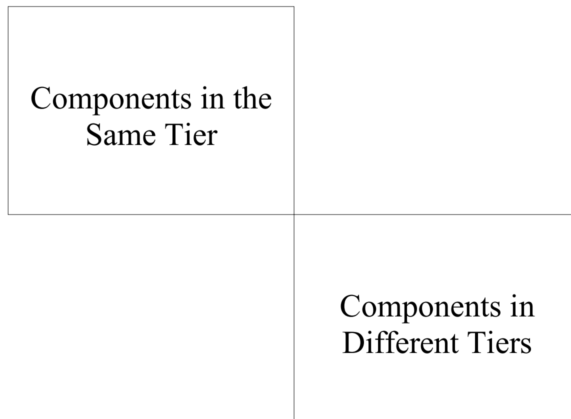
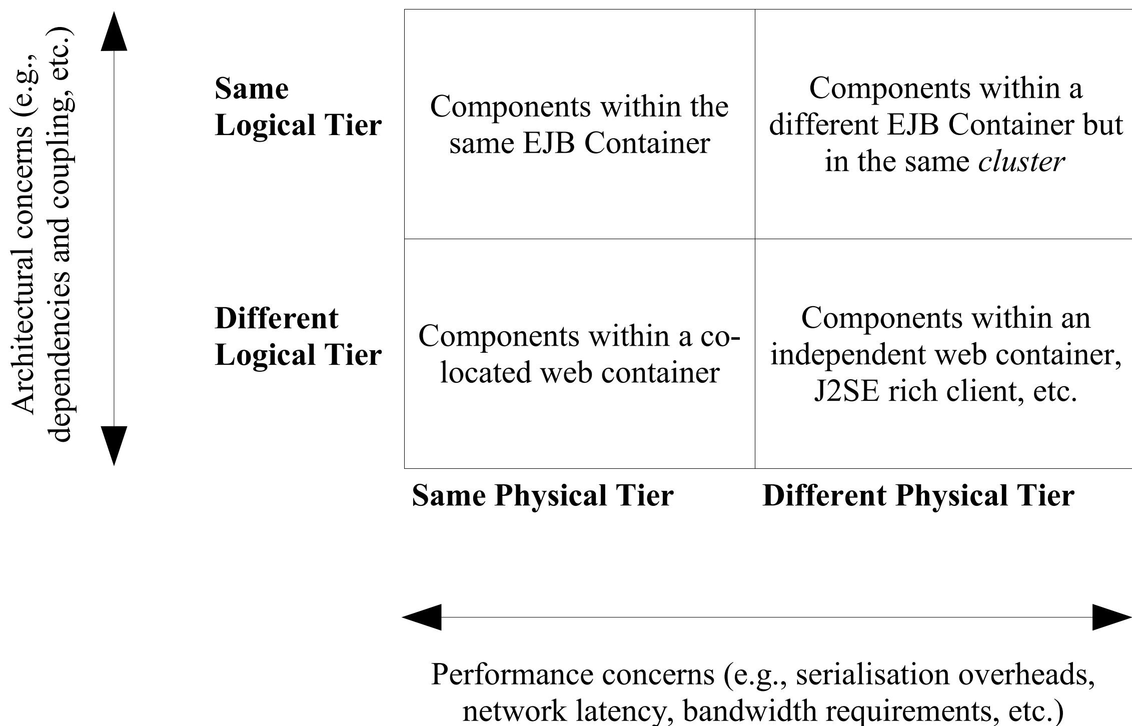


Figure 6 – The Real World View of Distributed Components

Tiers can be either logical or physical. Their implementations and implications are very different.



The architectural view of tiers is a logical one, and the concerns are around dependencies and tight coupling. This approach requires components in different logical tiers to be decoupled from each other. In other words, client components in a different tier must only receive references to objects that obey Transfer Object semantics, i.e., they are decoupled from the Domain Model and are understood not to represent it. This keeps clients shielded from changes in the Domain Model.

The performance view of tiers is physical, and the concerns are around serialisation and other networking issues. The performance overheads of serialisation are so high (reportedly around 3 orders of magnitude) that they strongly suggest the need for client components to share the same physical tier as far as possible (e.g., web containers and EJB containers co-located within the same J2EE server to share a common VM). To extract the maximum benefit from co-location, objects are passed by reference rather than by value, even though in-memory copying does not have the same overheads as serialisation.

It's obvious which view the EJB 3.0 spec committee takes, because their proposed solution is about passing Domain Model elements by reference whenever possible, and by value only when absolutely necessary. Their “detached entity” model is a way to painlessly switch between pass-by-value and pass-by-reference semantics. If the client component is physically co-located, then it receives a reference to the Entity itself. If the client component is physically remote, then it receives a reference to a “detached entity”. The client can only call the Session Bean's local interface from a physically local tier. If a client calls a Service method through a remote interface, then regardless of whether it is physically in the same tier or a different tier, it will only receive a detached entity – a copy. So there is no danger of mistaking the detached entity for a “live” one or vice-versa. Detached entities assume Transfer Object semantics.

All of this seems very neat, except for a few inconvenient implications:

1. The real-world costs of tight coupling: When the major way to exchange data with clients is by exposing the Domain Model itself, changes to client tier requirements will ripple through to the Domain Model, and changes to the Domain Model can break unknown clients who may have made assumptions about its structure. The application owners will have no visibility of such external dependencies when they change the internals of their application until someone screams. This model increases integration and testing effort, and makes changes difficult and costly. The concept of Service-Oriented Architectures emerged as a direct response to the proven high costs of tightly-coupled systems, but these lessons appear to have been forgotten. This is no Enterprise component model!

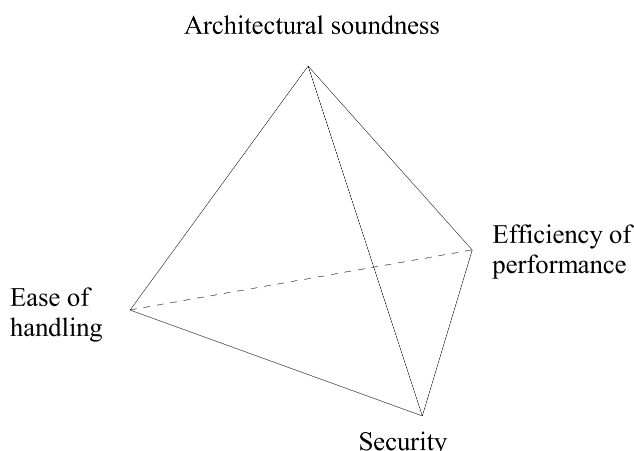
2. Network bandwidth overheads: Important Entities in real-life systems have a large number of attributes, and some of them can be quite large as well (e.g., Employee photographs in corporate HR applications, product photographs in e-commerce applications, etc.) What may have seemed a smart way to improve performance by avoiding copying can turn out to be nightmarishly expensive in cases where remote clients are involved, because detached entities holding hundreds of kilobytes of property information may have to be sent across networks, chewing up bandwidth. We cannot assume that clients of the EJB tier will only be web components in a co-located web container. Smart clients are becoming more popular, and remote web servers will never go away.

3. Security and privacy risks: Some Entity properties can be sensitive (e.g., personal details of employees, customer details covered by privacy policies, confidential product details, etc.). It is necessary to ensure that only properties relevant to a client component are sent to it. If a physically remote client requests an entity reference, and receives a “detached entity” with all its properties exposed, that can be a serious breach of privacy that can lead to litigation, or an unacceptable competitive risk. The EJB 3.0 model has no way to elegantly expose only what a client really requires to see, and therefore cannot support remote clients without exposing organisations to serious risk.

The Solution

We believe that the EJB architecture needs to address the four orthogonal concerns around data that we have raised, - architectural soundness, performance efficiency, security and ease of use, as illustrated by Figure 7. The EJB 3.0 spec has concentrated on only two of them (ease of use and performance), and has consequently done a bad job of the other two. Even performance in the remote case is likely to be far worse with that model.

Figure 7 – Four orthogonal concerns around the use of data



Solution description

In our model, a business domain is modelled using interfaces, as always. This covers processes as well as data. Hence Services and Conversations implement a Service Contract expressed through Business Interfaces (as everyone agrees), and *Entities implement the Domain Model expressed through Business Interfaces as well,* an observation that seems to have caused needless controversy.

There may be more than one Business Interface for an Entity, depending on the number of views of data within the domain. An example is an Employee Entity that appears as a Payee, SkilledResource, FacilitiesConsumer, ApplicationUser, etc., to different parts of an application. Each of these is a Business Interface, and declares its own set of properties. There could be overlapping properties between Business Interfaces, but these are resolved painlessly in the concrete implementation (If the primary design artifacts had been concrete classes, such flexible modelling would have been impossible). The Entity bean implements the superset of all these properties and provides transparent persistence in coordination with the container.

[We are not interested in the persistence of data here, only with the *use* of data. Persistence is assumed to be abstracted out of the equation by Entity Beans in coordination with the EJB container – The “Entity Bean Contract” promises clients of Entities (Services and Conversations) that they can remain oblivious to persistence concerns.]

Now the data requirements of **other** tiers have nothing to do with the domain model. Another tier of the application may want to deal with a set of properties in a convenient bundle, but it is a mistake to make an automatic association between this bundle and any part of the domain model.

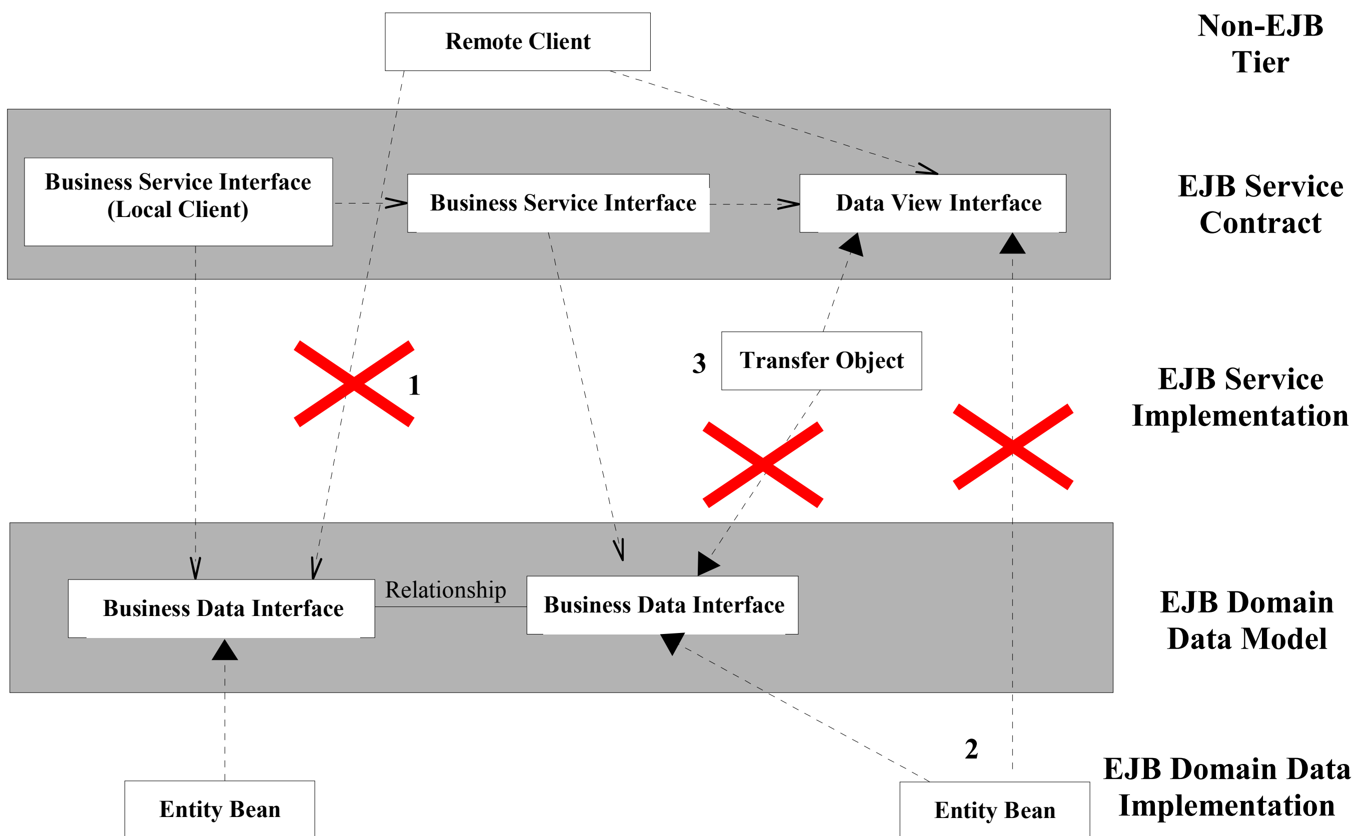
We've coined a new term for an external tier's data requirements – a "View Interface".

Entities do not implement View Interfaces because these interfaces are not intended to describe the domain. This means that some explicit translation must take place if properties are transferred between the Domain Model and any Transfer Object (an object that implements a View Interface).

A question that can arise here is whether the EJB tier needs to be responsible at all for View Interfaces or for translation between a View Interface and the Domain Model. If this is a client tier's data requirement, why should the EJB tier be responsible for it? The answer is that View Interfaces form part of the method signatures of Services, and are therefore part of the Service Contract. So even though they are not part of the Domain Model, the EJB tier as a whole is responsible for their stability and semantic integrity.

Figure 8 – Basic data-related principles of the proposed model

1. Entity references never cross physical tiers. If a physically remote client calls a Service method that returns an Entity reference, the container will throw an exception. Remote client dependencies on the Domain Model are thus made physically impossible by the container. (What about co-located "remote" clients? We'll discuss that later.)
2. Entities only implement Business Interfaces, never View Interfaces.
3. Transfer Objects only implement View Interfaces, never Business Interfaces.
4. Transfer Objects (although representations of data) are not part of the Domain Model but of the Service Contract. Therefore their creation and management are the responsibility of the Service Tier.
5. Even if a View Interface happens to be identical to a Business Interface, they must be physically distinct classes. Reuse of an interface class for these two separate purposes is prohibited.
6. Translation between Transfer Objects and Entities must be done in the Service Tier. Neither a Transfer Object nor an Entity must be aware of the other.



We can see 6 different situations that form a continuum. A good model needs to handle all of them in a clean manner, scaling gracefully with complexity (by making simple things simple and complex things possible):

Case 1: A Service method returns a reference to an Entity implementation class

Case 2: A Service method returns a Business Interface (which is implemented by an Entity)

Case 3: A Service method returns a View Interface (which happens to be identical to some Business Interface that is implemented by an Entity)

Case 4: A Service method returns a View Interface (all of whose method signatures happen to match corresponding ones in the Entity)

Case 5: A Service method returns a View Interface (*most* of whose method signatures match corresponding ones in the Entity)

Case 6: A Service method returns a View Interface (very few of whose method signatures match corresponding ones in the Entity)

In the previous incarnation of our model, we only handled Cases 2 and 6. We assumed that clients in other tiers would only need to see Business Interfaces. This was an example of tight coupling that we ourselves were guilty of. We also stated that where that was not the case, developers would hand-code their interfaces and implementation classes and do all the required translation by hand. That view was not sympathetic to ease-of-use concerns. We also implicitly assumed in that version of the model that Entities only implemented a single Business Interface, because we handled Case 1 by transparently cooking DTOs corresponding to that Business Interface. That was an invalid assumption. Finally, we relied on the developer to check if a reference to an object implemented the DTO marker interface or not. Although we avoided the overly broad use of local and remote interfaces, our model was dependent on the developer's diligence to work correctly. An absent-minded developer could leave a potential time-bomb ticking in an application by failing to check the local/remote semantics of an object.

Hence that model was clearly not sophisticated enough. We detail below how the new version of the model should work.

The major changes in the new model are:

1. The removal of the DTO marker interface (we propose a simpler mechanism that does not require the developer to perform a check in application code)

2. Prevention of Entity references being visible to other physical tiers

3. Automated translation between Entities and Transfer Objects wherever possible, with optional Translator components handling less straightforward property translation

Case 1: A Service method returns a reference to an Entity implementation class:

This is legitimate for clients within the EJB tier, because Services can pass Entity references to each other. But clients in other tiers must not call methods of Services that return Entity references, because they must remain insulated from the Domain Model.

Local clients receive a reference to the Entity itself. All updates made to it are automatically persisted. Remote clients get an exception if they try to call this method, because Entity references must not cross tiers.

Case 2: A Service method returns a Business Interface (which is implemented by an Entity):

This is very similar to the previous case, because the implementation class of a Business Interface is an Entity.

Local clients receive a reference to the Entity itself. All updates made to it are automatically persisted. Remote clients get an exception if they try to call this method, because Entity references must not cross tiers.

Case 3: A Service method returns a View Interface (which happens to be identical to some Business Interface that is implemented by an Entity):

This case is potentially very confusing and likely to lead to tightly coupled designs. If a single interface class is used to represent both a View Interface and a Business Interface, then the semantics of a Service method that returns this interface are unclear. Does the Service return a Transfer Object or an Entity? Nasty situations can result if the Service method returns a reference to one type of object but the client assumes a reference to the other type.

For this reason, our model requires that View Interfaces NEVER be used in situations requiring Business Interfaces, and vice-versa, even if they look identical in terms of their properties. Note to ease-of-use zealots: It is a trivial exercise to copy an interface file into another and name it differently.

It is thus the application that makes original/copy semantics explicit.

Now there is no danger that an Entity can be sent back by this method, because the View Interface is not implemented by any Entity and so no Entity can be cast to it. An explicit translation has to be done to meet the requirements of remote clients.

In our model, the EntityManager can provide such translation, acting as a Transfer Object Assembler. [We see the EntityManager not as an artifact of the Domain Model but as a Service Tier mechanism, because it provides services to Services.]

```
...
ProductView productView = (ProductView) entityManager.getView( productEntity,
                                                   ProductView.class);

return productView;
...
```

The developer does not have to define the Transfer Object implementation class. It is "cooked" by the EntityManager based on the View Interface definition and the relevant Entity properties are automatically copied across. Local clients receive a reference to a Transfer Object with the usual Transfer Object semantics (It must be explicitly passed back through another Service method to update the Domain Model). Remote clients actually get a reference to a *copy* of the Transfer Object, but the semantics of Transfer Objects ensures that this distinction is irrelevant. Just as before, the object must be explicitly passed back through another Service method to update the Domain Model. The EntityManager will do the update operation through reverse translation.

Note that the Transfer Object only holds values corresponding to properties in the View Interface. This makes the Transfer Object no larger than necessary, and also avoids exposing confidential properties unless the View Interface requires them.

Also note that the insistence on separate interfaces for Transfer Objects and Entities (even when they seem to be identical) avoids the need for local and remote interfaces on the Services themselves. The semantics of local and remote access to objects are captured in the interfaces themselves. Both clients and Services know what they are dealing with, and there is no ambiguity that can lead to problems.

<u>Case 4: A Service method returns a View Interface (all of whose method signatures happen to match corresponding ones in the Entity):</u>

This is very similar to the previous case. Although the Entity does not implement this interface, it is tempting to declare it as doing so, because the effort involved is trivial. After all, the Entity already implements all the methods in the View Interface.

Once again, our model requires that this NOT be done. View Interfaces and Business Interfaces are conceptually separate. Entities must not implement View Interfaces under any circumstances, even if it is trivial to declare them as doing so.

The translation must be done just as before through the EntityManager, with a simple method call. Transfer Objects modified by the remote client must be passed back to the EntityManager through another Service method to update the Entity through reverse translation.

As before, both local and remote clients get a reference to a Transfer Object, which must be explicitly passed back through another Service method to update the Domain Model.

<u>Case 5: A Service method returns a View Interface (*most* of whose method signatures match corresponding ones in the Entity):</u>

Translation in this case is not possible, and the EntityManager will throw an exception if called as before, because there are some properties in the View Interface that cannot be found in the Entity.

At this point, a straightforward solution might be to require the developer to define a Transfer Object implementing the View Interface and hand-code the entire translation. But human nature being what it is, there may be a temptation on the part of the developer to enhance the Entity to implement these additional properties, because then the translation becomes straightforward. This is obviously bad design, because it couples the Domain Model to the requirements of external tiers. How do we prevent it?

We believe we can prevent such abuses by making the correct approach easy to follow (so that principles of good design are never compromised), so we propose a Translator class written by the developer, defining mapping logic for *only* the methods that differ between the View Interface and the Domain

Model. The EntityManager uses the Translator to extract and transform values from the Domain Model (i.e., one or more Entities) before setting them in the Transfer Object that it creates. Other values that match those in the Domain Model (and are therefore implemented by the main Entity in question) are transparently copied without any effort on the part of the developer. The same principle is applied when updating the Domain Model later on from values in a modified Transfer Object. The Translator must define methods to do the reverse mapping as well. If either translation fails to find a required mapping method, the EntityManager will throw an exception.

```
...
ProductView productView = (ProductView) entityManager.getView( productEntity,
                                                               ProductView.class,
                                                               ProductViewTranslator.class);

return productView;
...
```

Both local and remote clients get a reference to a Transfer Object, which must be explicitly passed back through another Service method to update the Domain Model.

Translators are also useful to override methods with the same syntactic signature and provide different semantics. E.g., a View Interface's “getProductName()” method may actually correspond to

```
getProductName() + “ (“ + getProductCode() + “)”
```

in the Entity. The methods have the same signature (i.e., returning a String) but different meanings. The Translator can thus perform semantic translation as well.

[The simplest way to create a Translator (but one that is hard to test) is to define an abstract class that implements the View Interface and provides concrete implementations for only the methods that have no equivalent in the Entity. Then the container can conjure up a Transfer Object that is a concrete subclass of the abstract class, with the remaining property accessors implemented trivially. However, since testability outside the container is an element of ease of use, we believe that the Translator should be a concrete class. The design is a little less straightforward and requires more work on the part of the EntityManager, but it can be tested without a container.]

<u>Case 6: A Service method returns a View Interface (very few of whose method signatures match corresponding ones in the Entity):</u>

Once the View Interface begins to differ significantly from the domain model, it is unlikely that a simple Translator mechanism will suffice to perform the necessary mapping. There are probably deeper and more complex differences between the interfaces that need to be addressed through explicit application logic.

At this point, it is reasonable to expect that the developer will write a Transfer Object implementation of the View Interface by hand, and perform all required property translation in application code. Note that full hand-coding is only required in the most complex cases. Simple translations are handled trivially, and moderately complex ones may only require the exceptions to be hand-coded.

Both local and remote clients get a reference to a Transfer Object, which must be explicitly passed back through another Service method to update the Domain Model.

It seems likely that this design will work for object graphs as well.

Don't Transfer Objects violate encapsulation?

A frequently-voiced argument against Transfer Objects is that they "violate encapsulation". We disagree with this blanket statement. Yes, Transfer Objects *can* violate encapsulation, but only if they purport to represent the Domain Model itself. The EJB 3.0 detached entity model is a prime example of such a violation. If care is taken to explicitly decouple Transfer Objects from the Domain Model and move all inter-dependencies to a specialised translation component, then either of them can be modified without affecting the other. The Domain Model remains safely encapsulated from other tiers. This is exactly what we have done in our model. Transfer Objects implement View Interfaces and Entities implement Business Interfaces. These interfaces never share physical interface classes. Therefore Transfer Objects and Entities are explicitly decoupled.

We have also approached the ease-of-use issue through the use of automated mechanisms for translation wherever possible, with only exceptions handled through Translator classes. So developers generally do not have to create Transfer Objects by hand or write code to copy properties between Entities and Transfer Objects. It's usually a simple call to an EntityManager method.

What about performance?

Copying objects in memory (even deep copying) is far less expensive than serialisation, although we have no benchmark figures to prove it. Our model requires copying when moving between *logical* tiers, regardless of whether those tiers are *physically* separate or not. If those logical tiers are physically co-located, then the only overhead suffered is the copying overhead. If the logical tiers are also physically separate, then an extra serialisation overhead is suffered. Of course, there is no way to avoid the serialisation overhead for physically remote clients. Our model avoids exacerbating the serialisation and bandwidth problems by keeping the Transfer Object no larger than necessary, and this is possible because Transfer Objects are not detached Entities (which are omnibus collections of properties) but implementations of View Interfaces (which are defined subsets of those properties).

What about co-located "remote" clients?

We seem to have glossed over an important aspect of the tier concept, which we should address now. There is nothing in our model that prevents clients in other *logical* tiers from seeing the domain model directly if they happen to be co-located in the same *physical* tier. This can happen quite easily, because web and EJB components can be deployed to the same physical J2EE server. If Entity references are returned by Service methods (intended for consumption by other EJBs), there is nothing to prevent non-EJB clients from manipulating them directly from within a co-located web container. The container will not prevent such access by throwing exceptions. What happens then?

The application will work, and probably show good performance because there is no serialisation or even copying going on. The real issues arise if the web components are later moved to a physically separate tier. In that case, the application will stop working and throw exceptions wherever web components attempt to access Entities directly. This is good fail-safe behaviour, because the last thing we want is for the application to **seem** to work when the semantics of the object reference have actually changed. We want an early indication if the assumptions behind our design are no longer valid, and having the container throw an exception on attempted remote access of Entities is a safety mechanism.

So with this model, an application designer can take a deliberate decision in favour of performance by co-locating the web and EJB tiers and coupling the client view to the Domain Model. If the co-location

assumption is invalidated by a later move of the web components to a separate server, then the application owners are warned of the change by the fact that the application has stopped working, and they can choose at that stage to rewrite parts of the client to deal with Transfer Object semantics rather than Entity semantics. We don't recommend this approach, but it is a call for the designer to make given the constraints they face.

On the other hand, an application designer may choose to “do it right”, and keep the web component logically decoupled from the Domain Model through explicit View Interfaces and Transfer Objects, even when the web and EJB tiers are co-located. The application only suffers the relatively minor overhead of in-memory copying. That's a small price to pay for a decoupled architecture, which has so many other benefits. If the web component is later moved to another physical tier, no rewrite of the client is necessary, because the Transfer Object semantics assumed by the client remain the same.

What about ease-of-use and testability?

We take it as an intellectual challenge to come up with a model that is as easy to use as humanly possible. In spite of the largely favourable response to the earlier version of our model, many readers nevertheless expressed a preference for the EJB 3.0 model because they felt it to be simpler and more testable! This is a surprising view, because our model uses nothing but POJOs for implementation, with business interfaces providing a powerful design tool and a foolproof way to specify contracts between components, because concrete classes are guaranteed to implement their declared interfaces. At the same time, interfaces other than the identity interface are mostly optional, making components only as complex as they need to be. It is possible to create instances of the Entities, Services and Conversations outside the container with a simple “new” operation and to test those methods that do not have dependencies on container services.

In any case, we have revisited our component model and simplified it even further, removing restrictions that are not strictly necessary. Though these changes are not relevant to the Use of Data that is the topic of this document, we would like to take this opportunity to present our ease-of-use changes here:

1. There is no need for Entity Beans to declare the marker interface ContainerManagedPersistence anymore. All Entity Beans are assumed to be CMP by default, unless they implement the BeanManagedPersistence interface. If they do, they must also implement all the callback methods declared by that interface. CMP Entity Beans are free to implement all, none or just the creation subset of the Entity lifecycle callbacks by implementing the appropriate interface.

2. There is no need for Service and Conversation Beans to declare the marker interface ContainerManagedTransaction anymore. All Services and Conversations are CMT by default, unless they implement the BeanManagedTransaction marker interface. If they do, they get a reference to a FineGrainedTransactionController in their context. If they are CMT (the default option), they get a reference to a basic TransactionController in their context.

3. The interfaces that set the bean's context are no longer mandatory. They are now called Context Interfaces, and are only required to be implemented by beans that require their context, e.g., to access Service Locators or TransactionControllers. E.g., what was previously a mandatory interface called ServiceBean is now an optional one called ContextAwareService. If a Service Bean requires components that are only accessible through its context, it must implement ContextAwareService and the single method “setServiceContext()” declared in that interface. Beans that do not require the use of a context do not need to implement context interfaces.

All these measures are in line with our philosophy of keeping simple things simple, and increasing the complexity of the solution only when warranted, and only in proportion to the complexity of the problem itself.

Conclusion

We believe we have incorporated valid criticisms of the model we first presented and have addressed the many issues raised. We think we now have a good “story” that not only hangs together well from a software engineering perspective, but which also caters to performance, security and ease-of-use concerns.

To reiterate, our model is based on POJOs, with interfaces to define contracts between components. Interfaces provide modelling power, yet very few of them are mandatory. So the developer only implements the functionality that makes sense for a component. This keeps the entire application simple and understandable. Functionality that does not depend on container services can be unit-tested outside the container.

Significantly, we have also done away with the need for local and remote interfaces for beans, because our interface-based separation of Transfer Objects and Entities has provided a simpler and more intuitive way to distinguish pass-by-value and pass-by-reference semantics. In contrast, the EJB 3.0 spec continues to use the clunky 2.x mechanism which introduces an unnecessary programming construct into what should be straightforward business logic.

Most importantly (and this is why we stress it repeatedly), our model provides for simple and straightforward ways to handle simple cases, and imposes additional effort on the developer only in proportion to the complexity of the problem. It does not propose a uniformly heavyweight solution framework irrespective of problem complexity.

We therefore believe we have reconciled the hitherto contradictory requirements of architectural soundness, performance efficiency, ease-of-use and security, and have thus presented a challenge to the EJB 3.0 spec committee to better their design on all these aspects!

Ganesh Prasad & Rajat Taneja