Which Object Model to Expose: Transfer Object Model vs. Domain Object Model?

Karsten Klein, hybrid labs, Feb 2007
Version: 1.1

Introduction

In the past there have been very controversy discussions concerning the DTO (Data Transfer Object) pattern (also known as Transfer Object or – very unfortunate - Value Object pattern). Especially in the respective forums and blogs the pattern is assessed dauntlessly. The reasons for the controversy are manifold. The loudest opinions and contributions appear to be the result from sheer frustration [Codehaus1] and misuse of the pattern. Everybody involved in the discussion has of course good points, but in the end there are still no real answers – just extreme personal opinions.
The overall understanding of the pattern suffers from a vastly incomplete and unstructured discussion. Everybody wants to eliminate the need for DTOs [ChrisRichardson1], but nobody unravels the possibilities associated with the pattern.
As you may already conclude I'm an advocate of the DTO pattern, which is for several reasons I want to summarize in this document in order to eventually answer at least some of the questions.
In my eyes the only thing people have to get clear about is when this pattern can be applied successfully and where it just means a gigantic overhead that may even not be required.
However I also like to point out that there are several architectural decisions, which first of all have nothing to do directly with the pattern, but which may lead to it in order to avoid issues and problems that were not considered initially.

In short this article is meant to gather a complete list of arguments pro and con exposing transfer objects instead of the of domain objects towards any type of client i.e. the presentation layer of your application.

The Data Transfer Object Pattern

The DTO pattern description as it can be found on the web [Fowler1] is very incomplete. The pattern firstly has to be seen in the context of the full description in Martin Fowlers book [Fowler2], the associated discussion [Fowler3] and has to be further enhanced with considerations as they can be found in Sun's Core J2EE Patterns 'Transfer Object' [Sun1] and 'Transfer Object Assembler' [Sun2] – especially regarding the motivation aspects.
Moreover today's technologies (stepping away from the Sun Core J2EE patterns based on EJB 2.1 and ancestors) like object-relational mapping tools (such as Hibernate, Oracle TopLink or Sun's JDO), web services, service oriented architectures (SOA) and service data objects (SDO) have to be discussed in this context.
Additionally - accessing an underlying database - the focus also has to be drawn to the discussion of the pattern in context of transactions.
I advocate to clearly distinguish the terms transfer object and value object. Unfortunately, they are used as alias for each other – a related pattern, but – being very finical – they definitely are different concepts. However examining this in detail is not the objective of this document.

Terminology

I found that there is already a severe problem with terminology. I.e. the constructed persistence object model is often referred to as domain model (or the domain model is regarded to be the persistence model). This implies that the persistence model is a hundred percent congruent to the domain model. However, the technical realization of a domain model using a specific object-relational mapping technology and/or technology specific optimizations can impose additional obligations on the implementation of the domain objects. Especially, since these obligations are part of the resulting

object model, they are not hidden by Data Access Object (DAO) layer. This means that the choice of technology has an impact on the domain model. The result is a persistence technology flavored domain model, which may not be fully congruent to initial technically unflavored version. Therefore, not at last for the purpose of this document, I want to distinguish the terms and refer to the implementation of the domain modal as *technical domain object* model or even as *persistence object model* in order to stress the difference. The original unflavored term domain object model will have no direct realization in code and therefore is merely regarded as abstract concept.

Model	Description
Domain object model	Object model that describes your problem domain. It is used in the specification of your software to define and illustrate specific business and use cases. It is basically the result of a non-technical OOD.
Persistence object model or *Technical domain model*	Technical object model implementing the domain object model with focus on persistence in the database. The design may here be driven by the used persistence technology and may include considerations for optimizations.
Transfer Object Model	Object model used on the service layer for exposure towards the clients of the system (i.e. the presentation layer). The model supplements the exposed API.

Table 1: Terminology

It may happen that the transfer object model compared to the persistence model is much closer to the domain object model. And in fact the transfer object model is the model the clients (actors) of our system are dealing with. Here lies a fundamental opportunity. The transfer object model must no be anemic (that means that the transfer objects do not expose any particular behavior). In contrast is may contain transfer model specific behavior. This behavior must explicitly not match the domain logic defined on the internal technical domain model and can be completely different. Moreover, logic that is only internal can be simply isolated from the client.
Unfortunately when your model is exposed via web service your behavior is lost on the web service client stubs. Anemic models are considered an anti-pattern.

After this slight correction of terminology I'd like to rephrase the question that I'll try to answer in the subsequent paragraphs: What object model to expose: transfer object model or persistence object model?

To address this question the subsequent paragraphs will pick up on several technologies that are the usual suspects when building a best-of-breed software architecture. From every section I will extract advantages and disadvantages concerning the DTO pattern.

Object-Relational Mappers

From the developers of object-relational mappers such as hibernate you can get a very clear statement towards the transfer object pattern. It is an understatement to say that they seem to not like it very much. Understandably, since they have put a lot of effort in concepts like lazy-loading and detached objects. The question is how one can use these features, both with a certain degree of technology isolation and with making the best benefit out of them.

Lazy-Loading

The lazy-loading feature is not lost when using transfer objects. Only this feature is not directly used or exposed by the service API towards the presentation layer. It is merely used by service layer, where your application logic is supposed to be implemented and by the domain logic on technical level. The logic can benefit of these nice and good to have functionalities - while the client remains completely isolated from it by the transfer objects. In my eyes this is no curse, but a design goal: the

client should not be developed against a certain technology and its specific abilities. The assembly of the transfer objects - which again can be accounted to happen on the service layer - can naturally benefit from the lazy-loading feature as well.

Please note that when a accessing an object from the client, the client may also include some context information for loading the object from the database, such as to load an object with associations or without associations. The context information could be even more detailed and the assembly has to use this information during the transfer object assembly to tailor the DTO to the clients needs.

To put it in other words, lazy-loading is not a feature to use outside your service layer. If you do so it is a clear indicator that you are implementing business logic in the wrong place.

Detached Objects

Detached object are meant to prevent implicit database calls from the presentation layer using persistence objects directly. The implementation of detached objects today vastly depends on the used persistence technology. By exposing detached object to the clients you therefore subtly couple the client to a specific technology.

After a transaction to the persistence layer the state or completeness of the persistence object is not deterministic and dependents on the history of attributes accesses on the persistence object.

When using transfer objects the assembly happens within the transaction boundaries and therefore produces deterministic (especially no access-history-dependent) results.

Web Services

In conjunction with web services you may find yourself in the special situation of being forced by your chosen web service technology to provide transfer objects. This is especially true when your technical details of the persistence layer do not fit the technical requirements of the tools that are available for the web service technology of choice. The argument that today's mapping tools are good enough to provide a decent mapping is - at least from my experience and the experience in the development teams I have been working with - not quite holding true. Moreover I see it as a principal design goal to isolate technical concerns of the web service layer from the implementation of the persistence object model. Transfer objects are a very good way to achieve this isolation and even can provide an abstraction of the used technologies.

SOA

In service oriented architecture (SOA) you normally connect and orchestrate coarse-grained services, with defined and contracted interfaces. In addition to the already stated reasons above you here especially do not want a change on the persistence model to propagate through the complete application to your service layer. In order to isolate these concerns you would very naturally provide a transfer object model for this purpose.

SDO

Considerations towards exposing Service data objects (SDO) ad-hoc do not demand for a transfer object model. Nevertheless SDO may dictate to use a particular transfer object model in your application. [This paragraph is incomplete]. References [RobinRoos1]

General Considerations

Abstraction of Technology

As already mentioned in the previous paragraphs the use of transfer objects can isolate the concerns of the utilized technologies. I.e. in the web service scenario.

Access Control

Assuming a set of very specialized requirements for access control has to be implemented. It may occur to you that the features of the persistence layer or the underlying database are not meeting your requirements or needs to implement access control. I.e. if you want to be standard compliant

and use frameworks like JAAS for authorization. Based on these considerations you may implement authorization using an aspect oriented approach on the persistence object model.
In this scenario (and potentially others as well) the transfer objects would represent a facility to cache a 'personalized', access control filtered version of the persistence object.

Development Teams

Depending on the size of your development team and the number of interacting teams you may be forced to work with contracted interfaces and API specs. Again, as already mentioned in the SOA excursion the DTO pattern can help you to isolate your interfaces from changes in the persistence layer. Moreover in this special case the separation can support change management, by trickling changes up and down the layers in a controlled manner.

Using Code-Generators

When using code generators to create parts of your application the code for the transfer objects and the according assemblers may come for nothing, depending on what generator approach used.

Pro and Cons

This chapter summarizes in short all collected disadvantages and advantages. It can be used as checklist, whether you require DTOs or not.

Disadvantages

- Additional overhead for providing implementations for DTOs and DTO assemblers. The addition of an attribute has to be propagated through the assemblers to the persistence model. Also there is a computational overhead in the conversion (using the assemblers) of the objects.
- DTO specific service interfaces and implementations (service adapters) are required. These implementations do nothing more than converting objects using the assemblers. In addition there is again a computational overhead in the conversion and by adding yet another layer of indirection to the call stack.

Advantages

- DTOs do not have to directly match the persistent object model. They provide a decoupling that is resolved by the DTO Assemblers and the service adapters. This decoupling may be of significant advantage when the interface of the API is evolving and the database remains unchanged. Furthermore the database structure may also evolve (or the whole persistent layer may be exchanged, without the API to change). → Isolation from interface and/or database evolution.
- The persistence layer (such as EJBs, hibernate entities, etc) may expose some technical details, which normally would not be exposed on the API level. In this case DTOs provide isolation of technical details. These technical details may be tied to the respective technology being used as persistence layer (hibernate, jdbc, etc), which means again that DTOs allow to abstract from the underlying persistence technology. → Isolation from technical details of system internals and the underlying persistence technology.
- Isolation of logic. Two models enable to provide two different sets of domain logic. The logic on the technical domain model can be isolated. The transfer object model logic becomes part of the external interface → Isolation of logic, Preserved and complete interfaces.
- DTOs can aggregate several calls to the database. → Separation of layers in the application. A field access to the used object in e.g. the web gui, does not result in a jump across all layers to the persistence level; → Isolation of application layers. This advantage looses weight in combination with detached object however the next argument has to be regarded simultaneously
- DTOs have deterministic content. That means they merely depend on the context information provided at the service invocation time. This argument does especially not apply for detached objects. In this case the content of the detached object depends on the history of the

persistence object. Especially when parts and pieces are already in the cache of the Session/PersistenceManager the content of the detached object may differ.

- An architecture using DTOs can ensure that there are no further calls to the database operating on the DTOs. All loading happens inside the transaction calling the service. This further ensures short transaction and less locking in the database.
- The DTO assemblers can isolate the persistence object model from prohibited modifications. Sometimes nothing can prevent the client from modifying an object the client is not supposed to change (e.g. by faking a web service request). The assembler may be the place to detect such things and prohibit such modifications by throwing an appropriate exception.
- Application or architecture may require that the client MUST not know about technical details of the technical domain objects. Technical infrastructure used on technical domain object level can be hidden towards the clients.
- DTOs can implement the required interfaces for an agreed service layer. The persistent entity may not be able (due to technical constraints) to implement the interface. I'd even say it should not be concerned with such domain specific considerations and should mainly focus on the technical implementation (which may be problematic enough).
- SOA requirement: isolate interface from actual implementation. The implementation is only a technical concern. DTOs provide this isolation. The service interface and the actual implementation can also be bundled separately. I.e. the interface bundle with the DTO can be delivered independently to other teams or partners without tying these 'client' to technical details or even to provide them with business logic exposed on the internal object model that has either not been stabilized yet or actually is not part of the agreed API. The problem of choosing an appropriate DTO granularity is not in scope of this document.
- Separation of Business logic for client and server side. Server side business logic (application level is implemented on a service level (in the J2EE context a session bean) interacting with the technical domain model and the appropriate DAOs. Object specific logic is to be implemented on the technical model itself. The client in contrary does not see the persistence model at all. It operates strictly only on the DTOs and may implement logic on this (integration) level.
- DTOs can be easily generated. Even when incorporating a specialized mapping.
- DTOs and the DTO Assembler add another level of control. E.g. attributes that are only of technical intrinsic meaning are intentionally not exposed on the API (DTO) level.
- Depending on your authorization mechanism DTOs provide caching on permission filtered results. Personalized DTO; shared (cacheable) Domain Object.
- DTOs may provide additional attributes that are filled during the access. These attributes may include computed values or data accessed through via additional service (external) calls.

References

[Codehaus1]	http://blogs.codehaus.org/people/tirsen/archives/ 000859_data_transfer_objects_makes_me_sick.html
Fowler1]	http://www.martinfowler.com/eaaCatalog/dataTransferObject.html
[Fowler2]	Patterns of Enterprise Application Architecture, Addison Wesley
[Fowler3]	http://martinfowler.com/bliki/LocalDTO.html
[Fowler4]	http://www.martinfowler.com/bliki/AnemicDomainModel.html
[Sun1]	http://java.sun.com/blueprints/corej2eepatterns/Patterns/ TransferObject.html
[Sun2]	http://java.sun.com/blueprints/corej2eepatterns/Patterns/ TransferObjectAssembler.html
[RobinRoos1]	http://www.jdocentral.com/jdo_commentary_robinroos_6.html
[ChisRichardson1]	Pojos in Action, Manning