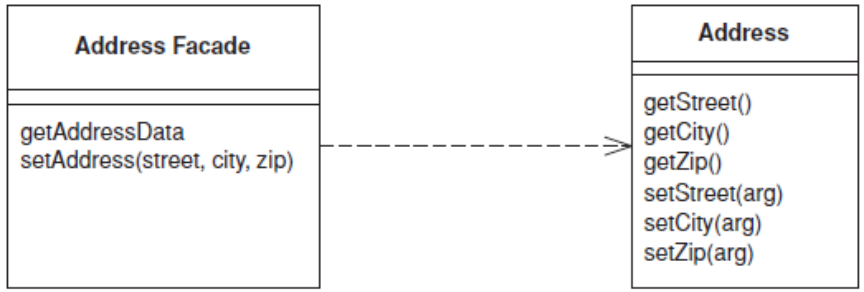
SOEN 387: Web-Based Enterprise Application Design

Chapter 15. Distribution Patterns

Remote Facade

Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.



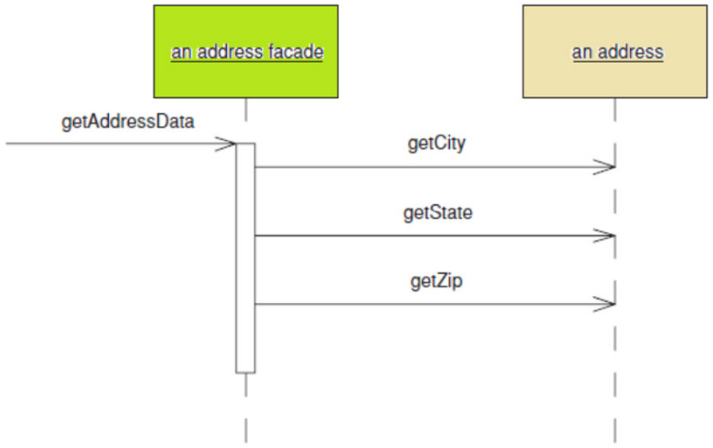
In an object-oriented model, you do best with small objects that have small methods. This gives you lots of opportunity for control and substitution of behavior, and to use good intention revealing naming to make an application easier to understand. One of the consequences of such fine-grained behavior is that there's usually a great deal of interaction between objects, and that interaction usually requires lots of method invocations.

As a result any object that is intended to be used as a remote objects needs a coarse-grained interface that minimizes the number of calls needed to get something done.

Rather than ask for an order and its order lines individually, you need to access and update the order and order lines in a single call.
This affects your entire object structure. You give up the clear intention and fine-grained control you get with small objects and small methods.

A *Remote Facade* is a coarse-grained facade over a web of fine-grained objects. None of the fine-grained objects have a remote interface, and the *Remote Facade* contains no domain logic.

All the *Remote Facade* does is translate coarse-grained methods onto the underlying fine-grained objects.



One call to a facade causes several calls from the facade to the domain object

In a simple case, like an address object, a *Remote Facade* replaces all the getting and setting methods of the regular address object with one getter and one setter, often referred to as **bulk accessors**.

When a client calls a bulk setter, the address facade reads the data from the setting method and calls the individual accessors on the real address object and does nothing more.

In a more complex case a single *Remote Facade* may act as a remote gateway for many fine-grained objects.

For example, an order facade may be used to get and update information for an order, all its order lines, and maybe some customer data as well.

In transferring information in bulk like this, you need it to be in a form that can easily move over the wire.

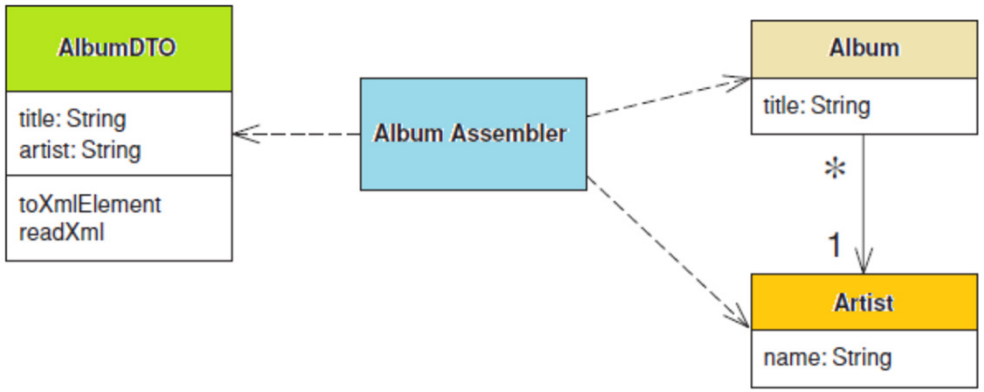
If your fine-grained classes are present on both sides of the connection and they are serializable, you can transfer them directly by making a copy.

In this case a *getAddressData* method creates a copy of the original address object. The *setAddressData* receives an address object and uses it to update the actual address object's data.

Often you can not do this, however. You may not want to duplicate your domain classes on multiple processes, or it may be difficult to serialize a segment of a domain model due to its complicated relationship structure.
The client may not want the whole model but just a simplified subset of it. In these cases it makes sense to use a *Data Transfer Object* as the basis of the transfer.

Data Transfer Object

An object that carries data between processes in order to reduce the number of method calls.



When you're working with a remote interface, such as *Remote Facade*, each call to it is expensive. As a result you need to reduce the number of calls, and that means that you need to transfer more data with each call.

The solution is to create a *Data Transfer Object* that can hold all the data for the call. It needs to be serializable to go across the connection. Usually an assembler is used on the server side to transfer data between the DTO and any domain objects.

Data Transfer Object allows you to move several pieces of information over a network in a single call—which is essential for distributed systems.

Whenever a remote object needs some data, it asks for a suitable *Data Transfer Object*.

The *Data Transfer Object* will usually carries much more data than what the remote object requested, but it should carry all the data the remote object will need for a while.

Due to the latency costs of remote calls, its better to err on the side of sending too much data than have to make multiple calls.

A single *Data Transfer Object* usually contains more than just a single server object. It aggregates data from all the server objects that the remote object is likely to want data from.

Thus, if a remote object requests data about an order object, the returned *Data Transfer Object* will contain data from the order, the customer, the line items, the products on the line items, the delivery information—any many others.

You can not usually transfer objects from a *Domain Model* . This is because the objects are usually connected in a complex structure that is difficult, if not impossible, to serialize.
Also you usually don not want the domain object classes on the client, which is tantamount to copying the whole *Domain Model* there. Instead, you have to transfer a simplified form of the data from the domain objects.

The fields in a *Data Transfer Object* are fairly simple, being primitives, simple classes like strings and dates, or other *Data Transfer Objects*.
Any structure between data transfer objects should be a simple graph structure—normally a hierarchy—as opposed to the more complicated graph structures that you see in a *Domain Model* .

Keep these simple attributes because they have to be serializable and they need to be understood by both sides of the wire. As a result the *Data Transfer Object* classes and any classes they reference must be present on both sides.

It makes sense to design the *Data Transfer Object* around the needs of a particular client.
That is why you often see them corresponding to Web pages or GUI screens. You may also see multiple *Data Transfer Objects* for an order, depending on the particular screen.
Of course, if different presentations require similar data, then it makes sense to use a single *Data Transfer Object* to handle them all.

Serializing the *Data Transfer Object* Other than simple getters and setters, the *Data Transfer Object* is also usually responsible for serializing itself into some format that will go over the wire. Which format depends on what is on either side of the connection, what can run over the connection itself, and how easy the serialization is.

One of the most common issues you face with Data Transfer Object is whether to use a text or a binary serialization form.

Text serializations are easy to read to learn what's being communicated. XML is popular because you can easily get tools to create and parse XML documents.

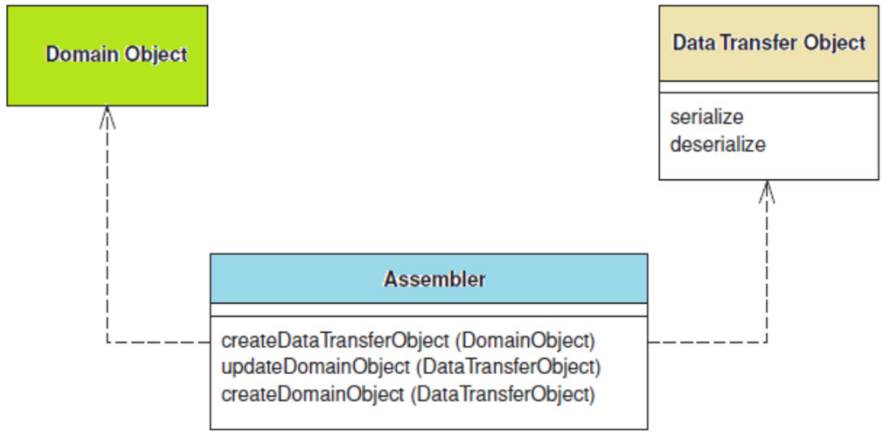
The big disadvantages with text are that it needs more bandwidth to send the same data (something particularly true of XML) and there's often a performance penalty, which can be quite significant.

Assembling a *Data Transfer Object* from Domain Objects A *Data Transfer Object* does not know about how to connect with domain objects. This is because it should be deployed on both sides of the connection.

For that reason *Data Transfer Object should not* be dependent on the domain object. Nor the domain objects to be dependent of the *Data Transfer Object* since the *Data Transfer Object* structure will change.

As a general rule, we want to keep the domain model independent of the external interfaces.

Make a separate assembler object responsible for creating a *Data Transfer Object* from the domain model and updating the model from it .
The assembler is an example of a *Mapper* in that it maps between the *Data Transfer Object* and the domain objects.



An assembler object can keep the domain model and the data transfer objects independent of each other.

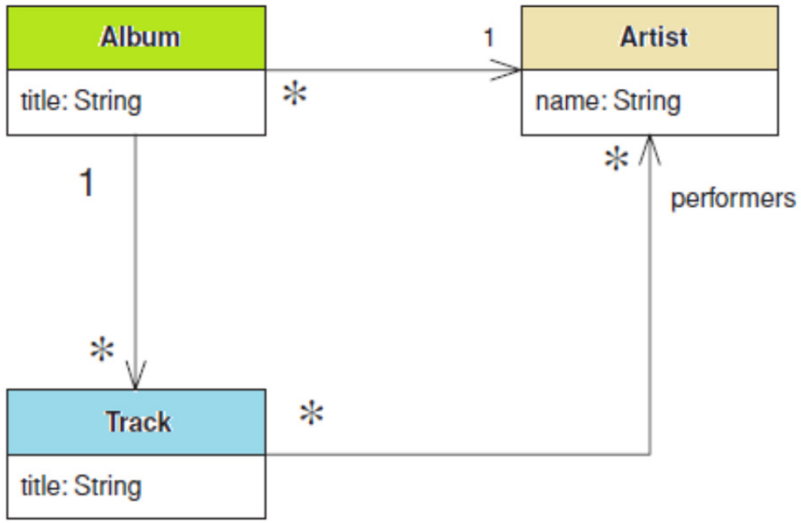
Use a *Data Transfer Object* whenever you need to transfer multiple items of data between two processes in a single method call.

Another common purpose for a *Data Transfer Object* is to act as a common source of data for various components in different layers. Each component makes some changes to the *Data Transfer Object* and then passes it on to the next layer.

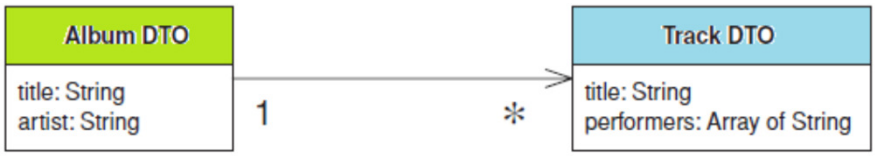
Example: Transferring Information About Albums (Java)

Consider the diagrams below: There are two data transfer objects present, one for the album and one for each track.

In this case we don not need one for the artist, as all the data is present on one of the other two. we only have the track as a transfer object because there are several tracks in the album and each one can contain more than one data item.



A class diagram of artists and albums.



A class diagram of data transfer objects.

Here is the code to write a *Data Transfer Object* from the domain model. The assembler is called by whatever object is handling the remote interface, such as a *Remote Facade*.

```
class AlbumAssembler...
public AlbumDTO writeDTO(Album subject) {
          AlbumDTO result = new AlbumDTO();
          result.setTitle(subject.getTitle());
          result.setArtist(subject.getArtist().getName());
          writeTracks(result, subject);
          return result;
}
private void writeTracks(AlbumDTO result, Album subject) {
          List newTracks = new ArrayList();
          Iterator it = subject.getTracks().iterator();
          while (it.hasNext()) {
                    TrackDTO newDTO = new TrackDTO();
                    Track thisTrack = (Track) it.next();
                    newDTO.setTitle(thisTrack.getTitle());
                    writePerformers(newDTO, thisTrack);
                    newTracks.add(newDTO);
}
          result.setTracks((TrackDTO[]) newTracks.toArray(new TrackDTO[0]));
}
private void writePerformers(TrackDTO dto, Track subject) {
          List result = new ArrayList();
          Iterator it = subject.getPerformers().iterator();
          while (it.hasNext()) {
                    Artist each = (Artist) it.next();
                    result.add(each.getName());
          }
          dto.setPerformers((String[]) result.toArray(new String[0]));
}
```