Design Modeling

Topics in This Chapter

- Creating a Design Model
- Design Guidelines
- Mapping a Domain Model to a Design Model
- Designing Data Access Objects, Transfer Objects, and Business Rule Objects

Chapter 2

In designing a complex J2EE business application, a design model is a necessity.

A design model is a detailed specification of the objects and relationships within an application, as well as its behavior. To understand the requirements originating from the business and to preserve the ability to trace these requirements to the application's implementation and back, it is clearly beneficial to keep the design model and domain model in sync. This chapter describes how to derive a design model from a common domain model, preserving this needed synchronization.

Also included in this chapter are guidelines for designing the integration tier of a multi-tiered J2EE application intended to integrate with legacy systems and other data sources. These guidelines are presented in terms and concepts associated with design models as presented in this chapter. Design patterns such as Transfer Objects and Data Access Objects [*Core J2EE Patterns, D. Alur, J.Crupi, D. Malks, Prentice-Hall 2001*] are used as the building blocks from which the integration tier is designed.[1]

1. These objects are referred to as Value Objects in the first edition of *Core J2EE Patterns*, but have been renamed Transfer Objects in the second edition.

2.1 Creating a Design Model

A design model is an object model that describes the realization of use cases, and it serves as an abstraction of the implementation model and its source code [*The Unified Software Development Process, I. Jacobsson, G. Booch, J.* Rumbaugh, *Addison-Wesley 1999*]. A design model consists of the following artifacts:

- **A class diagram:** This diagram contains the implementation view of the entities in the domain model. Each object in the design model should, ideally, be exactly traceable to one or more entities in the domain model. This property ensures that the requirements, as specified in use cases containing entities defined in the domain model, are realized by corresponding classes in the design model. The design model also contains non-core business classes such as persistent storage and security management classes.
- **Use case realizations:** Use case realizations are described in collaboration diagrams that use the class diagram to identify the objects that participate in the collaboration diagrams.

If the class diagram of the design model is created as a derivative of the class diagram of a domain model, each class in the design model traces to a corresponding class in the domain model. There can be one-to-one, one-to-many, and many-to-many relationships between design classes and domain classes.

Because it should be possible to implement a domain model in more than one way, the direction of this class tracing should normally be only *from* the design model *to* the domain model. Furthermore, keeping the domain model consistent with updates in the derived design models is impractical.

The traceability of a design model to the domain model aids IT architects and application designers by providing a realization of the use cases that closely corresponds to the business entities defined in the domain model. In other words, there is no confusion over domain model entities used to describe business use cases, since their corresponding design classes also exist in the design model.

Mapping a Domain Model to a Design Model

An entity defined in the domain model is represented as a Transfer Object and a Data Access Object in the design model (see Figure 2.1).

For each domain model class having the stereotype <<entity>>, the design model contains a corresponding Transfer Object and possibly also a Data Access Object. Each domain model class having the stereotype <<utility>> is mapped to a

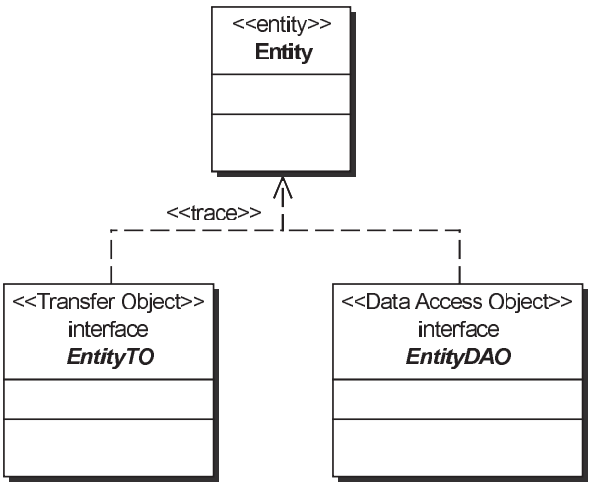

Figure 2.1 The realization of an entity class

Supporting Object class in the design model. Classes stereotyped <<type>> are mapped to enumerated type classes.

The design model class diagram consists of UML interfaces, classes, and associations. The following sections describe in detail procedures by which entities and relations in a domain model are mapped to elements in a design model. Note that the procedures apply to both the common domain model and application-specific domain models.

Entities

Each domain model class having stereotype <<entity>> is mapped to a corresponding interface in the design model having stereotype <<Transfer Object>>. Each attribute of this class is mapped to public `get<AttributeName>` and `set<AttributeName>` methods in the corresponding interface.

Types

As previously stated, a domain model class having stereotype <<type>> should always inherit from a base class that names the entity to which the type belongs. In the design model, this is represented by a class having stereotype <<Supporting Object>> with the class name `<Entity>Type` (the name of the base entity followed by the word Type). The subclasses having stereotype <<type>> are then mapped to constant attributes in the Supporting Object class.

Figure 2.2 shows an example in which different Product types are mapped to the Supporting Object class ProductType, which contains the constant object attributes INFORMATION_PRODUCT and FINANCIAL_PRODUCT.

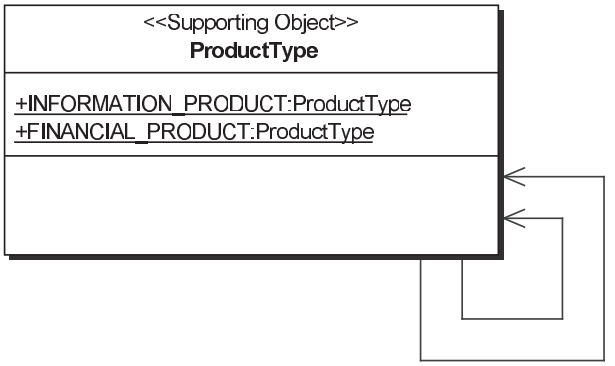

Figure 2.2 The ProductType class

When a type has subtypes, the type and subtypes should be collapsed into a flat structure of values.

The Transfer Object interface representing the base entity in the domain model should have both a `get<Entity>Type` method and a `set<Entity>Type` method in the interface definition, where the get method returns one of the constant objects of the `<Entity>Type` class, and the set method takes a constant object of the `<Entity>Type` class as in-parameter.

Utilities

In the design model, a domain model utility is represented as a class having stereotype <<Supporting Object>>.

Associations

In the design model, an association between two domain model entities is mapped to a `get<AssociationName>` method and a `set<AssociationName>` method in the Transfer Object Interfaces corresponding to the domain model entities. For example, a Party having the association CustomerRole to a Customer would be represented as a class Party having `getCustomerRole` and `setCustomerRole` methods. Similarly, the Customer class would be designed to have `getPartyRole` and `setPartyRole` methods.

To avoid retrieving too much data when reading a Transfer Object from persistent store, associations between entities can be represented with identifiers rather than with direct object references to Transfer Objects. (This concept is discussed in greater detail later in this chapter.) With this method, there is no type information directly available from an association. When implementing the application from the design model, therefore, the domain model must be used to indicate from which Data Access Object a referenced Transfer Object should be retrieved.

Another option is to create Business Rule Objects that, based on a Transfer Object and an association name, return the Data Access Objects that then retrieve the correct Transfer Object instances.

Association Classes

Some associations consist of an association class. Depending on whether or not the association class has an identifier, it can be represented in the design model as either a Transfer Object interface or a Supporting Object class. A one-to-many association between two entities consisting of an association class could be mapped in two ways:

1. As a one-to-one association between the first Transfer Object and the association class, which contains a one-to-many association to the second Transfer Object
2. As a one-to-many association between the first Transfer Object and the association class, which contains a one-to-one association to the second Transfer Object

The following guideline is helpful in choosing one of these approaches: If the association class represents membership in a group, the first type of mapping should be used; otherwise the second mapping should be used. The following examples illustrate this concept:

> Example 1: A Party has a colleague role to many other Parties. Since these all work for the same employer, the colleague role is represented through an Employer class having its own identifier and name and so on. Each Party has a one-to-tone relationship with the Employer class, while the Employer class has a one-to-many relationship with the Party class.
>
> Example 2: A Party has a supplier role to many other Parties. The Parties, representing customers in this case, each have a unique customer ID in the supplier's own records. That is, this ID is not part of the Party definition but a property of the association class Customer. Since the customers in this case are not members of a group. The relationships between the Party having the supplier role and the Parties having the customer role are represented with a number of one-to-many associations from the Party class to the Customer class, as well as with a one-to-one association from Customer class to the Party class.

Summary of Guidelines

Table 2.1 summarizes guidelines that should be used in creating a design model class diagram from a domain model:

Table 2.1 Mapping guidelines

Domain model element	Design model representation	Stereotype	Restrictions	Comment
Entity	Interface	<<Value Object>>	Must have a unique identifier Must be associated with a Data Access Object	
Type	Class	<<Supporting Object>>	Types are mapped to constants	Add `get<Entity>Type` and `set<Entity>Type` to Transfer Object interface
Association	Association, possibly aggregate	none		
Association Class	Interface	<<Value Object>>	Must have a unique identifier Must be associated with a Data Access Object	Entity one-to-one to Association Class and one-to-many to Entity or Entity one-to-many to Association Class and one-to-one to Entity
Association Class	Class	<<Supporting Object>>	Must not have a unique identifier	Entity one-to-one to Association Class and one-to-many to Entity or Entity one-to-many to Association Class and one-to-one to Entity
Utility	Class	<<Supporting Object>>		

Design Model Mapping Example

Figure 2.3 shows an example of a domain to design model mapping

Additional Design Model Classes

Once the straight mapping from domain model to design model has been performed, some additional design model classes need to be added.

Composite Transfer Objects

A Composite Transfer Object is a representation of an association between two or more Transfer Objects. Because associations between entities can be represented with identifiers rather than with direct object references to Transfer Objects, a separate object for maintaining these associations is needed. This object is represented in the class diagram as a class having the stereotype <<Composite Transfer Object>>. By designing relationships through Composite Transfer Objects, an application designer is free to implement only those associations needed by the application, thus avoiding unnecessary object instantiations and possibly circular references. Guidelines for designing Composite Transfer Objects are discussed later in this chapter.

Transfer Object Classes

Each Transfer Object interface that is not extended by another Transfer Object interface is complemented by a Transfer Object class that implements the Transfer Object interface and all of its extended interfaces. The implementation class should have the name <ValueObjectName>Impl and be stereotyped <<Transfer Object>>. In addition, the following properties should be applied to Transfer Objects:

- A Transfer Object should represent data retrieved from one or more records obtained from one or more data sources.
- Each Transfer Object must have a primary key (although there may be more than one key) that identifies the record in the data source.
- Transfer Objects should not contain other Transfer Objects.
- Transfer Objects must never contain or reference Composite Transfer Objects; instead, whenever possible, Transfer Objects should reference other Transfer Objects using their primary keys.
- Transfer Objects should have no advanced business logic; this rule promotes reuse and simplifies the architecture. In particular, because Transfer Objects often are transported from one tier to another, logic related to communication with other parts of the system should be avoided.

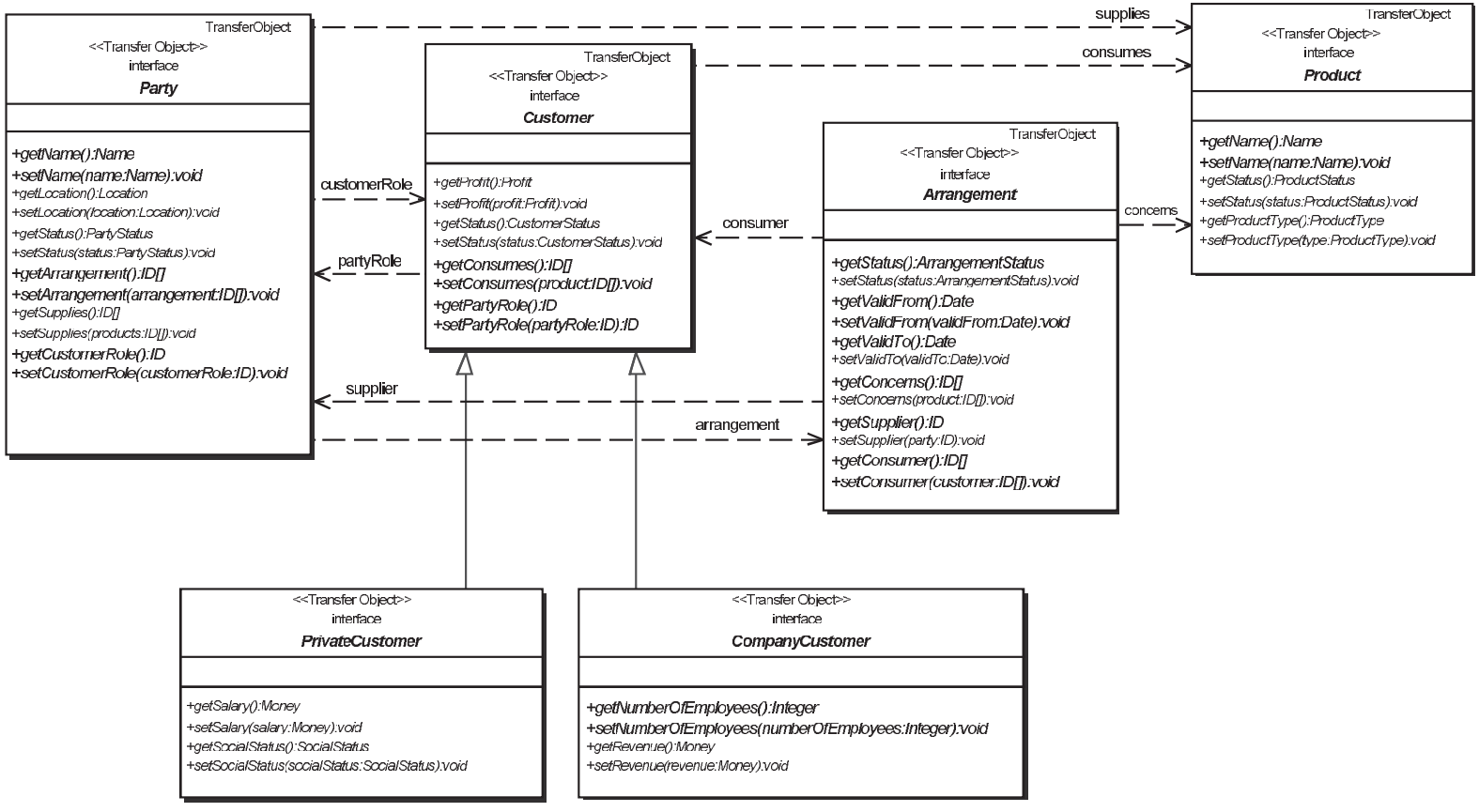

Figure 2.3 An Entity to Transfer Object mapping

- It should be possible to export Transfer Objects to XML format.
- Transfer Objects should be designed as JavaBeans™, providing a get method and set method for each attribute.

Supporting Objects

Supporting Objects constitute attributes in Transfer Objects. A Supporting Object is usually persistent but has no primary key. A Supporting Object must never contain or have a reference to a Transfer Object or a Composite Transfer Object. Supporting Objects should be implemented as a class stereotyped <<Supporting Object>>.

Business Rule Objects

The rules found in the business rule catalog of the common domain model can be represented in the design model as interfaces having stereotype <<Business Rule>>. These objects can, for example, verify the contents of a Transfer Object or of a Supporting Object. They can also be used to perform calculations based on the attribute values of a Value Object or Supporting Object. In order to be portable and reusable, a Business Rule Object must not have any side effects, such as manipulating the attribute values of a Value Object or Supporting Object. Furthermore, a Business Rule Object must not invoke methods on remote objects or communicate with external resources such as databases. Guidelines for designing Business Rule Objects are discussed later in this chapter.

Data Access Objects

For each Transfer Object class, there should be a corresponding Data Access Object interface, with the name <ValueObjectName>DAO and having the stereotype <<Data Access Object>>. Data Access Objects represent the integration tier that connects J2EE applications with a legacy system where Transfer Object data is stored. Guidelines for designing Data Access Objects are discussed later in this chapter.

Optimizations

Because the design model is intended to be implemented, optimizations such as factoring out common attributes into an abstract super class are encouraged. For example, the ID attribute common for all entities could be factored out and placed in a separate Transfer Object interface and corresponding abstract base class.

Another example is the case in which an entity has more than one subclass containing attributes and identifier. In this case, each subclass must be mapped to a Transfer Object interface and a corresponding Transfer Object class, where each class implements the base Transfer Object interface. In order to avoid replication, the base Transfer Object interface can be complemented by an `Abstract<base Transfer Object Interface Name>` class, which is extended by the Transfer

Object implementations. Note, however, that the abstract Transfer Object must not have a separate Data Access Object implementation, since it is not possible to create instances of the abstract Transfer Object.

Transfer and Data Access Object Class Diagram Examples

Figure 2.4 shows a class diagram containing Transfer Object implementation classes. Note that the Transfer Object interfaces as well as methods and attributes have been omitted in this figure.:

Figure 2.5 shows a class diagram containing Data Access Objects. Note that methods have been omitted in this figure.

2.2 Design Guidelines

In addition to accurately reflecting the domain model of the enterprise, an application design model must also address application characteristics that could cause problems at run time, as well as when maintaining and modifying the application. The following three design guidelines help to avoid major problems that usually appear in applications with legacy system integration unless addressed during the design phase

- **Performance:** Minimize access to the persistent store to mitigate the performance impact of accessing legacy systems.
- **Flexibility and reuse:** Design the integration tier so that component reuse and re-implementation become easy and straightforward.
- **Integration:** Overcome the impedance mismatch between an object-oriented data representation and its corresponding representation in a legacy system.

These guidelines are based on the assumption that the application is structured in tiers. Note, however, that only software tiers are considered here; in this context, an application tier can reside on any physical tier. For example, all application tiers can reside on the same physical tier, or one application tier can span multiple physical tiers. Figure 2.6 shows the application tiers as defined in *Core J2EE Patterns* [*D. Alur, J. Crupi, D. Malks, Prentice-Hall 2001*].

From this figure, it is clear that the role played by the integration tier of an application is to isolate the business tier from the resource tier, in order to make the business tier resilient and minimize the impact of changes in the resource tier.

In creating a design model from a domain model, the core business entities, types, utilities, and relations of the domain model are mapped to elements such as Transfer

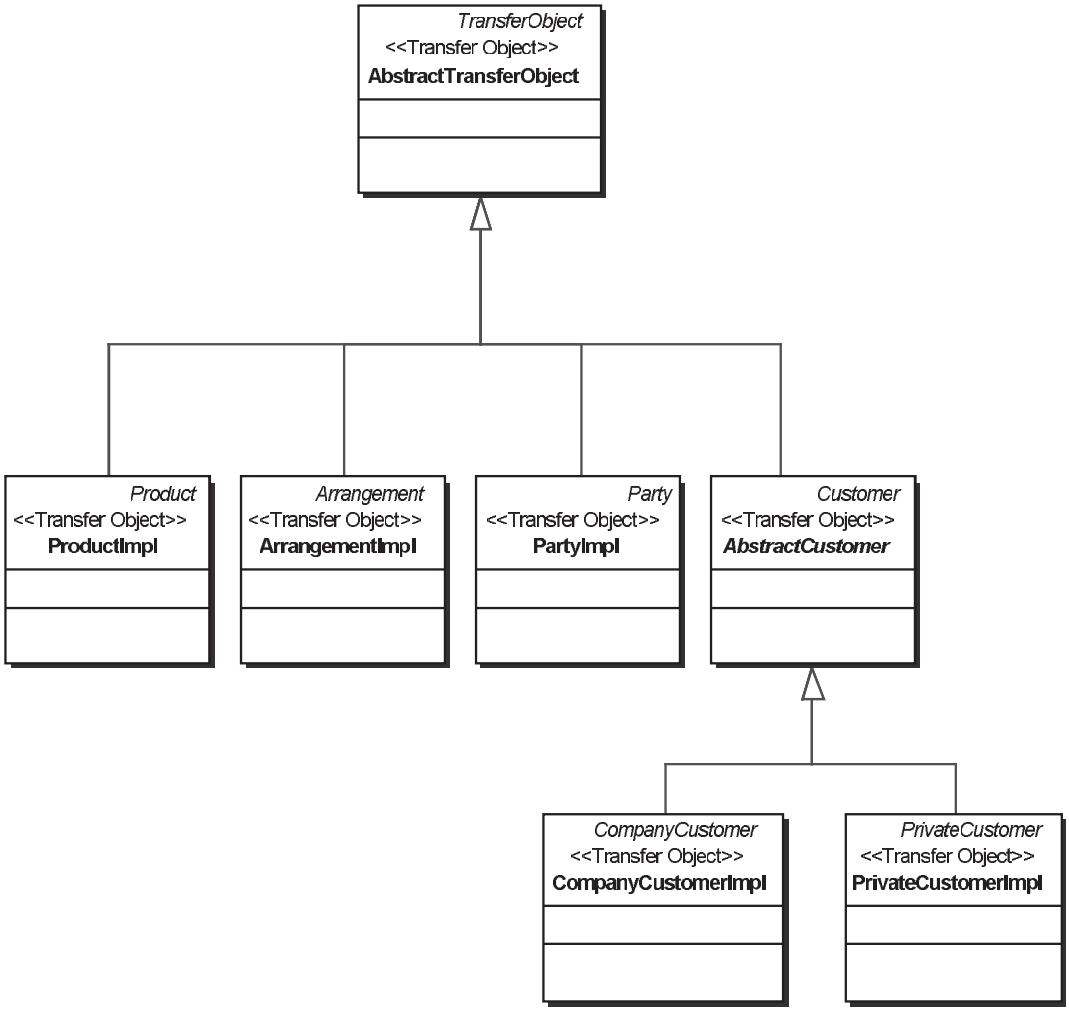

Figure 2.4 Transfer Object implementation classes

Objects and Data Access Objects, from which the integration tier is designed. The following sections discuss these concepts and how to apply them to an integration tier design.

Managing Entity-to-Entity Relations

Overly fine-grained object-oriented applications tend to have a performance impact on large-scale distributed systems. Increased component granularity provides a

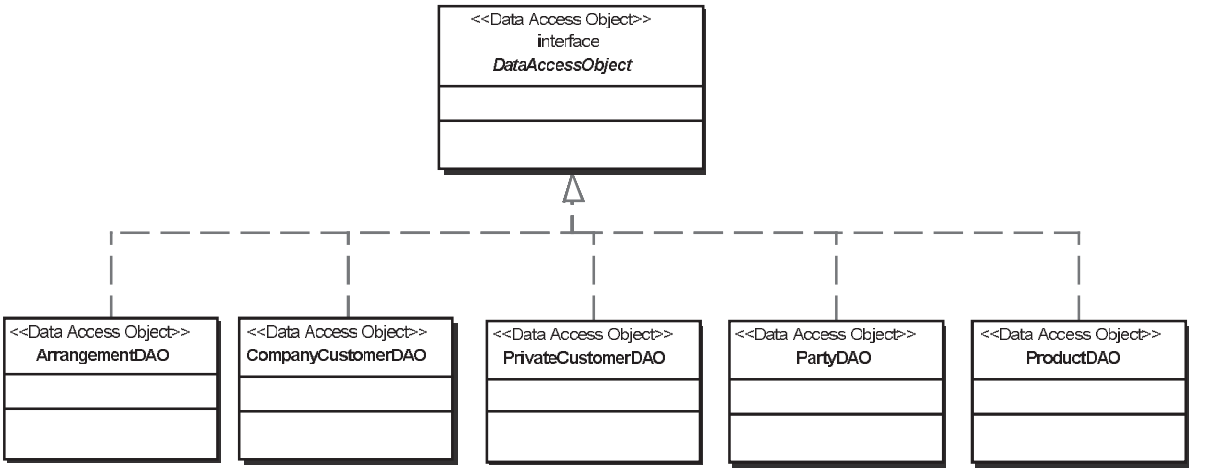

Figure 2.5 Data Access Objects

means of controlling that impact. Communication and data access, the main bottlenecks in any distributed system, are normally the focal points for any optimizations made in the application architecture.

In order to minimize communication between the business tier and the resource tier, two principles are essential:

- The business tier should access only large chunks of data, that is, whole objects rather than single attributes.
- The business tier should resolve only those entity-to-entity relations that are required by the application.

Applications often need to retrieve collections of business entities from the persistent store. For instance, an application might provide a method to retrieve a list of all the Arrangements of a Party (see Figure 1.1). The representation of Transfer Object collections is an area in which performance can be improved by careful design. How should the Transfer Objects Party and Arrangement be designed to represent the relationship between them?

Using the traditional object-oriented approach, the class representing Party would be designed to have a collection of references to Arrangement objects. However, this approach sometimes leads to complex situations in which data not actually required by the application is retrieved from persistent store, or in which circular references must be addressed (see Figure 2.7).

To avoid these problems, Transfer Objects should not contain object references to other Transfer Objects. Associations should instead be represented as object

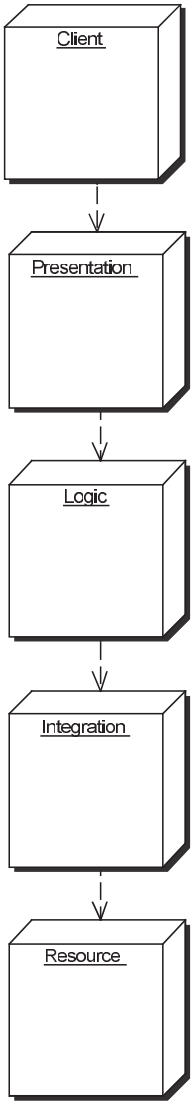

Figure 2.6 A five-tiered architecture

identities or primary keys. To continue our example, the Party contains an array of identifiers representing primary keys of Arrangements (see Figure 2.8).

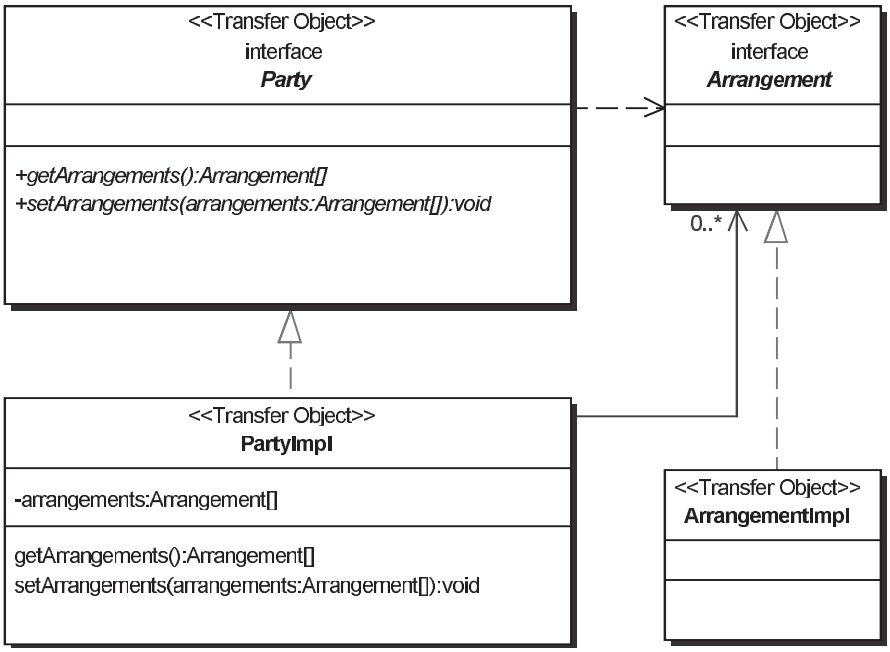

Figure 2.7 Transfer Object containing object references

Composite Transfer Objects

A Composite Transfer Object is a representation of an association between two or more Transfer Objects. In our example, the association between a Party and its Arrangements can be represented by a class called, for example, PartyArrangements, that contains an attribute of type Party and a second attribute consisting of a sequence of Arrangement objects (see Figure Figure 2.9).

The purpose of representing associations in separate objects is to prevent the application from reading the entire database when instantiating a Transfer Object. For example, in the domain model depicted in Figure 1.1, a Party has associations to Arrangements that have associations to Products that have associations to Customers that have associations to a Party. The risk of reading the entire database is apparent, since domain models often tend to relate everything to everything.

By using Composite Transfer Objects, an application can resolve only the needed associations (in our example, only the association between Party and Arrangement), leaving the remaining associations unresolved (in our example, this would mean that the association between Arrangement and Product would remain unresolved, among others).

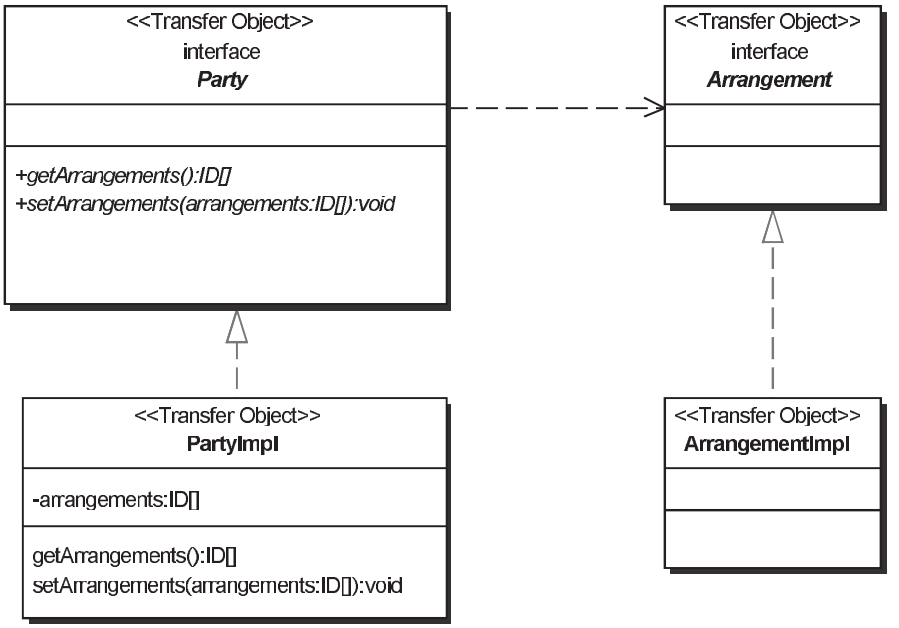

Figure 2.8 Transfer Objects containing ID attributes

This approach is sometimes referred to as lazy evaluation. The motivation for this design can also be found in *Design Patterns* [*E. Gamma et al., Addison-Wesley 1994*], in which it is shown that decoupling objects allows varying their interactions without having to subclass or modify them. Furthermore, when the domain model changes, only the mediator class (that is, the Composite Transfer Object) must be redesigned.

The disadvantage with this approach is that unresolved associations lose the type information normally contained in object references. For example, leaving the association from Arrangement to Product unresolved means that when the application wants to access this association, the type information is not available. The application itself, therefore, must be aware of which Transfer Object class (for example, Product) to retrieve from persistent store in order to resolve the association. This domain knowledge can, however, be contained in a Business Rule Object, as discussed later in this chapter.

Design for Flexibility and Reuse

Making the integration tier components flexible and reusable is a common problem. Component-driven design is the key technique in obtaining flexibility and reuse. Any component should be designed according to the following principles:

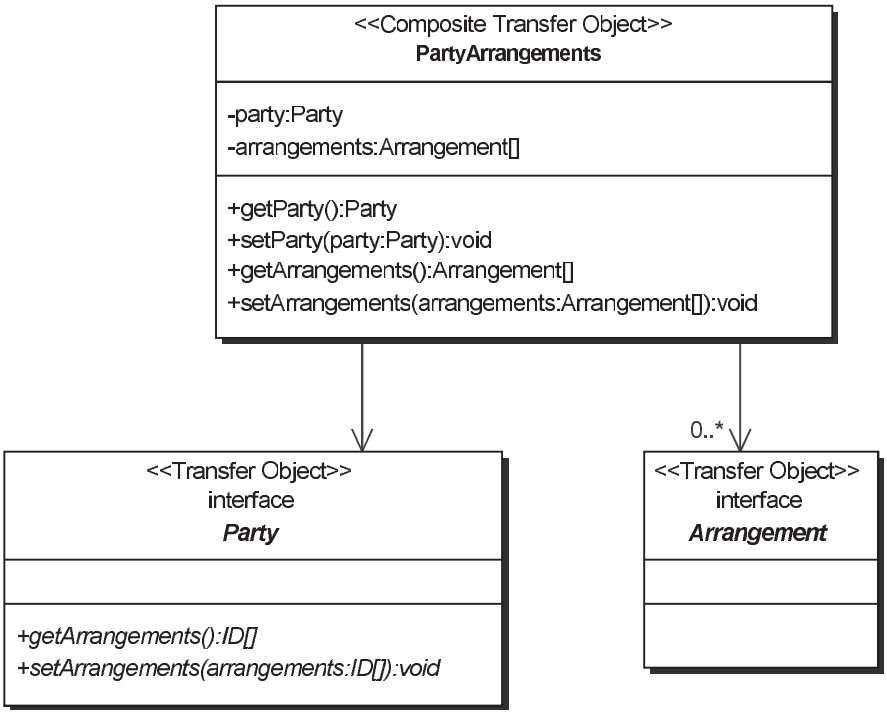

Figure 2.9 A Composite Transfer Object containing Party and Arrangements

- A well-defined interface that hides the component's implementation, providing flexibility
- Service-driven design, in which the component is designed to provide a service and does not care to whom this service is provided, thereby promoting reuse

Applying these principles to the integration tier gives it a good chance of becoming truly flexible and reusable. Furthermore, in order to make the integration tier's components usable by any device or application, they should be designed to make no assumptions about the presentation format of data retrieved from the resource tier.

Composite Transfer Objects, Transfer Objects, and Data Access Objects provide excellent means for achieving flexibility and reusability.

The use of Composite Transfer Objects, as discussed earlier, allows the associations between Transfer Objects to be created dynamically and managed in a flexible manner. By using Transfer Objects to represent data, a normalized data representation is provided within the business tier and toward the presentation tier.

(Note that this use of Transfer Objects is an internal representation only. For external representation—for example, toward other, possibly non-Java applications—it must be possible to transform data from the Transfer Object format to other formats, XML being the prevailing choice.)

Data Access Objects integrate the business tier with the resource tier, providing the means for the business tier to access persistent data without knowledge of the implementation of the persistent store. In this way, the business tier is shielded from changes in underlying resources.

Designing the integration tier for flexibility also requires that logic concerning entities, relationships, and data access is located in the appropriate components. For example, logic concerning a calculation or a constraint based on the attributes of a single entity, such as a Party, could very well be located in a Transfer Object. Transfer Objects, however, should never directly access a persistent store or a remote service. If a Transfer Object is to access a persistent store or a remote service directly, it must contain logic that depends on the infrastructure of a particular application, which could make the Transfer Object less reusable in the context of another application running on a different infrastructure.

Generally, in order to make the application as flexible as possible, it is advisable to locate business logic outside plain data carriers such as Transfer Objects, by using Business Rule Objects, for example, as discussed in the following section.

Business Rule Objects

A Business Rule Object is a design model element that represents enterprise business logic. For example, an application can call a sequence of Business Rule Objects in a certain order to verify that the business rules of a particular method are followed.

Business Rule Objects should operate primarily on objects that are passed to it as method parameters. As with Transfer Objects, therefore, and for the same reasons, Business Rule Objects should not normally communicate with persistent stores or with remote services. It is possible, however, for a Business Rule Object to operate on Transfer Objects, as well as on Supporting Objects and Composite Transfer Objects.

A Validator is the interface of a Business Rule Object that is used to validate the contents of an object according to constraints set on the application. For example, this interface is used by the application to validate input parameters passed to the application, as well as the contents of Transfer Objects returned from Data Access Objects.

Other types of Business Rule Objects, given a Transfer Object and a relationship name, can return a Transfer Object class. For example, a Business Rule Object could be given a Party and the relationship name arrangements and return an Arrangement

class object. The most important considerations in designing Business Rule Objects is that they should be both simple to modify (as this modifies the behavior of an entire application or set of applications) and self-contained, in order to be reusable in all tiers of an applications and between different applications.

For example, a Validator interface can provide a single method called validate, which returns true or false, while the class that implements the Validator interface in turn contains a constructor that accepts one or more Transfer Objects and/or Supporting Objects. Complex business rules can then be composed of a number of Validators combined in a logical expression.

For example, assuming Validators A, B, and C, these could be combined as follows:

```
boolean d = A.validate() & (B.validate() | C.validate());
```

Now consider the business event "Withdraw an amount from an account" and let A, B, and C be as follows:

- A.validate: Return true if the customer is the owner of the account
- B.validate: Return true if the balance of the account is equal to or greater than the amount
- C.validate: Return true if the customer has a credit tied to the account that is equal to or greater than the amount

Then define the composite rule this way: True if the customer is the owner of the account AND if the balance of the account is equal to or greater than the amount OR if the customer has a credit tied to the account that is equal to or greater than the amount.

The Transfer Objects involved in this rule could be, for example:

- Rule A: Customer and Account
- Rule B: Account
- Rule C: Product and Account

This results in the object seen in Figure 2.10:

Note that A, B, and C are self-contained and reusable in any tier and in any application that deals with Customers, Accounts, and Products. By changing the implementation of either one of the Validators, the behavior of the application can be changed without any major recoding.

Business Rule Objects can be implemented through the use of a commercial rule engine or by implementing them as normal Java objects.

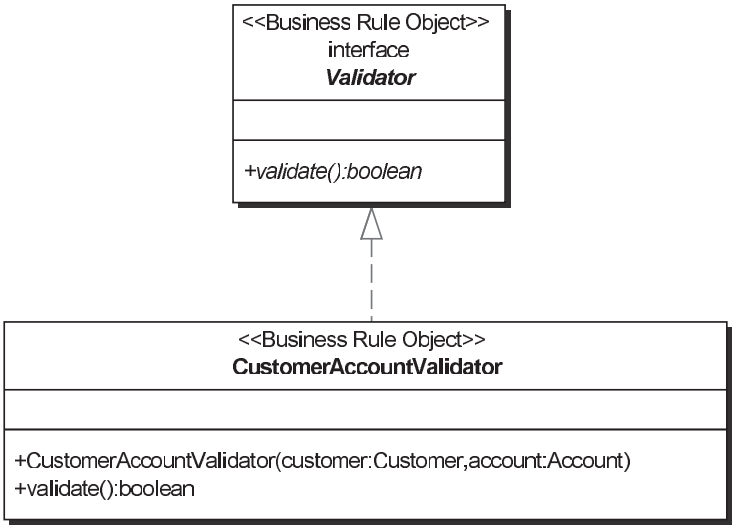

Figure 2.10 A CustomerAccountValidator object

Managing the Impedance Mismatch

As object-oriented applications are developed on top of legacy systems and data sources that are not object-oriented, the impedance mismatch between the object-oriented structuring of enterprise data (entities) and the corresponding legacy or relational structuring becomes apparent. Most enterprises have an increasing need to become less dependent on their legacy systems. Therefore, domain models created for new applications without taking legacy systems into consideration can deviate dramatically from the legacy systems' structuring of enterprise data. Unless this object-oriented structuring is adhered to, however, J2EE applications risk becoming the legacy systems of the 21st century.

The Enterprise JavaBeans (EJB™) specification provides entity beans in combination with container-managed persistence as the primary method for encapsulating data sources. But there are situations when this approach cannot be taken. As discussed earlier, entity beans introduce a potential scalability and cache inconsistency problem; for some types of applications, therefore, entity beans are not feasible. When it comes to container-managed persistence, there are a number of situations in which straightforward object-to-persistent data mappings are not possible. The following section discusses this in more detail.

When Container-Managed Persistence Cannot Be Used

With container-managed persistence (CMP), the EJB container manages data source access and mappings between transient entities (represented as entity beans) and

persistent entities (represented as persistent records). In order for an application to use CMP, the following conditions must be met:

1. CMP is used only with entity beans.
2. The EJB Container Provider provides a tool for mapping entity beans to persistent records.

As noted, there are a number of situations in which CMP cannot be used and they are explained in this list:

- More than one entity is mapped to a single legacy system service. (In this case, a CMP implementation might call the same service more than once.)
- An attribute value might need to be converted. (For example, a value returned might need to be converted from uppercase to lowercase or be internationalized.)An attribute of an entity is mapped to a part of a field in a record. (This means that the field must be parsed and that business rules might need to be applied.)
- An entity is mapped to multiple legacy system services. (For example, a list service might be followed by a read service based on the values of one of the rows returned in the list.)
- An entity is mapped to more than one data source, and it is necessary to pick attribute values from different sources and combine them into a single Transfer Object.
- The data source does not provide support for distributed transactions.

The Data Access Object pattern, in combination with Bean-Managed Persistence (BMP), provides a solution for all of these situations. The next chapter describes this solution in detail.

Pitfalls of Caching Persistent Data

Because data access is a potential bottleneck in any distributed system, it is tempting to use application data caching to reduce the number of transactions with the resource tier. In a thousands-of-transactions-per-second application, a cache can help significantly by avoiding the trip to the resource tier whenever possible. It is important, however, to be aware of two potential pitfalls in the use of caches in distributed systems:

- An application data cache can introduce a single point of failure. Where the application's data caches cannot be replicated in a transactional way without using a database, a particular application cache is useful only if all requests arrive at the same machine. That cache thus becomes a single point of failure. Transactions to

synchronize distributed caches are needed; without them, a cache replica might be updated by a client before being resynchronized with the other caches.

- An application cache will become inconsistent with the persistent store as soon as the application is no longer the only way to access the persistent store. For example, if the persistent store is part of a legacy system, there will be numerous other applications accessing the persistent store through other channels. Hence the cache will constantly become invalid, requiring frequent refresh calls to the legacy system.

Applying this knowledge to an application can guide us in choosing the right granularity for data access, and can also be helpful in choosing between session beans and entity beans.

Entity beans are meant to serve as a cache for persistent data that is accessible by many simultaneous requests, existing only in a single active instance for a particular business entity instance. If they existed in more than one instance, as noted above, the application could have a cache consistency problem.

Ideally, entity bean granularity would match the domain model entity granularity; as a cache, however, entity beans suffer from the problems described earlier. Furthermore, entity beans cannot be treated as rows in a relational database, since entity-to-entity bean relations and the joining of multiple entity beans into a single object create extreme performance demands.

When combined, and given the single point of failure risk, these drawbacks makes entity beans inappropriate for applications that need to serve large numbers of simultaneous requests to the same entity. Furthermore, they are inappropriate for applications that need to integrate with legacy systems, due to the cache inconsistency problem caused by other applications.

In order to avoid these pitfalls, it is essential to remember that an application data cache often cannot be used for shared data using general-purpose J2EE application servers. Instead, data access must normally be performed in large chunks, fetching as much as possible each time. The use of the Data Access Object pattern in combination with coarse-grained legacy system calls provides a means of combining the fine-grained domain model with a coarse-grained legacy system representation of core business entities.

Data Access Objects

The Data Access Object pattern addresses the impedance mismatch discussed in the previous section by isolating the business tier components from the actual interfaces to the legacy systems. Since the data passed to and returned from these interfaces are normally not formatted according to the domain model used by modern applications,

the most important task of the Data Access Object is to reformat the enterprise data into Transfer Objects and vice versa. Furthermore, by designing Data Access Objects and Transfer Objects based on entities of the domain model rather than on the information structure of the legacy systems, the impedance mismatch can be isolated within the integration tier, not affecting the remainder of the application.

If we view the Data Access Object as having two parts, an interface and an adapter [*Design Patterns, E. Gamma et. al., Addison-Wesley 1994*], the way in which the Data Access Object pattern provides a mechanism for application portability across legacy systems becomes straightforward. By keeping the interface while replacing the adapter, a business tier component can access data from legacy system A today and legacy system B tomorrow. Furthermore, standardizing the interface of the Data Access Object allows the final decoupling of the application from the information structure of the legacy system.

It is important to realize that the interface of the Data Access Object must not provide any services beyond what would be expected from any data source. Normally, these services are creating, reading, updating, deleting, and searching. By keeping strictly to this set of services, the ability of the Data Access Object to adapt to new data sources can be maximized.

Figure 2.11 shows a session bean consisting of two Data Access Objects. Each Data Access Object manages Transfer Object instances of a different class, each containing data from three different data sources. These data sources can be included in distributed transactions controlled from the session bean.

All persistent stores can be encapsulated within a Data Access Object. The purpose of a Data Access Object is to provide methods for creating, reading, updating, and deleting Transfer Objects (not Composite Transfer Objects or Supporting Objects). Distributed transactions can include multiple invocations of Data Access Objects. A Data Access Object must be dedicated to one and only one Transfer Object class—Party, for example. Since Transfer Objects can consist of data from more than one data source, a Data Access Object is able to merge data from one or more data sources into a single Transfer Object.

The interface of a Data Access Object should provide the following methods:

- `create(newVo:TransferObject):ID`
  Create a new record in the persistent store. Return a primary key to that record.
- `read(primaryKey:ID):TransferObject`
  Read a record given a primary key. An exception should be raised if there is no matching record.
- `update(updatedVo:TransferObject):void`
  Update a record with the data contained in the provided Transfer

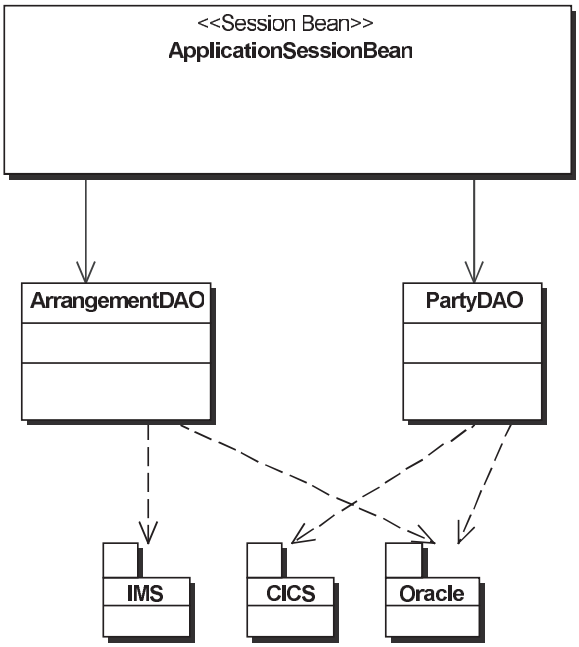

Figure 2.11 A Session bean, two Data Access Objects, and three data sources

Object. An exception should be raised if there is no record containing the primary key provided in the Transfer Object.

- `delete(primaryKey:ID):void`
  Delete a record given a primary key. An exception should be raised if there is no matching record.

- `find(constraint:TransferObject, relOp:Comparator):Iterator`
  Find Transfer Objects having attribute values matching the values provided in the constraint. An object implementing the java.util.Comparator interface is provided to compare the constraint with the Value Objects retrieved from the data source. The Comparator object can be implemented specifically for each Transfer Object to provide a method for comparing two objects for equality, greater than, or lesser than.

- `find(query:Query):Iterator`
  Return an Iterator object containing all records matching the query. The query must evaluate a set of Transfer Objects of the same class as managed by the Data Access Object. An iterator containing zero records is a valid result and should not raise an exception.

In cases in which some methods cannot be implemented due to limitations in the persistent store, the method can either raise an exception or provide a default implementation.

It is important to note that all Transfer Object instances passed to or returned from these methods in a particular Data Access Object must be of the same class—the Transfer Object class that is managed by the Data Access Object. However, because all Data Access Objects provide a common set of operations, it is possible to provide a common base interface for all Data Access Objects (see Figure 2.12).

As discussed in Chapter 1, since each Transfer Object class is to be managed by a corresponding Data Access Object class, a typical application will consist of a large number of different Data Access Object classes. A good approach to managing a large set of classes is through an Abstract Factory [*Design Patterns, E. Gamma et. al., Addison-Wesley 1994*]. The concept of a Data Access Object Factory is discussed in the next chapter.

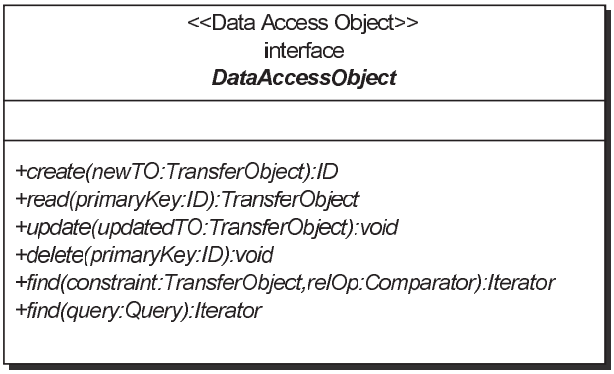

Figure 2.12 The Data Access Object interface

Through the use of adapters, Data Access Objects provide application portability across legacy systems. This approach makes it possible for an enterprise to implement adapters based on the common domain model for all its legacy systems. Since these adapters adhere to the standardized Data Access Object interface, it is straightforward to implement J2EE applications based on the common domain model independently from the implementation of the adapters. This decoupling provides the means of reusing the integration tier from application to application. Furthermore, since the design model used to implement the integration tier is derived from the common domain model, a traceable, reusable software component can be utilized throughout the enterprise, with any legacy system.

2.3 Summary

This chapter has explained how to map an enterprise domain model to a design model that can be used to implement the integration tier of a J2EE application. Through this straightforward mapping, design elements such as Transfer Objects and Data Access Objects can be directly traced to core business concepts such as Party, Arrangement, and Product.

The design guidelines and patterns for J2EE applications presented in this chapter have shown how Transfer Objects, Composite Transfer Objects, Business Rule Objects, and Data Access Objects provide the building blocks for a design model that addresses the fundamental characteristics of an application, such as performance and flexibility. In addition, these guidelines address the challenging task of bridging the impedance mismatch between an object-oriented design model and the enterprise data representation of a legacy system.

The Data Access Object pattern is the key to overcoming this impedance mismatch. However, the mapping from an object-oriented enterprise data model to a legacy system service model is complex, and this complexity is increased by the addition of distributed transactions involving legacy system services that cannot provide two-phase commit functionality to a J2EE application. This issue is addressed in Chapter 3, and techniques for combining Data Access Objects with the J2EE Connector Architecture are presented.

Implementing a design model derived from the common domain model allows the creation of a core, reusable component for the enterprise.