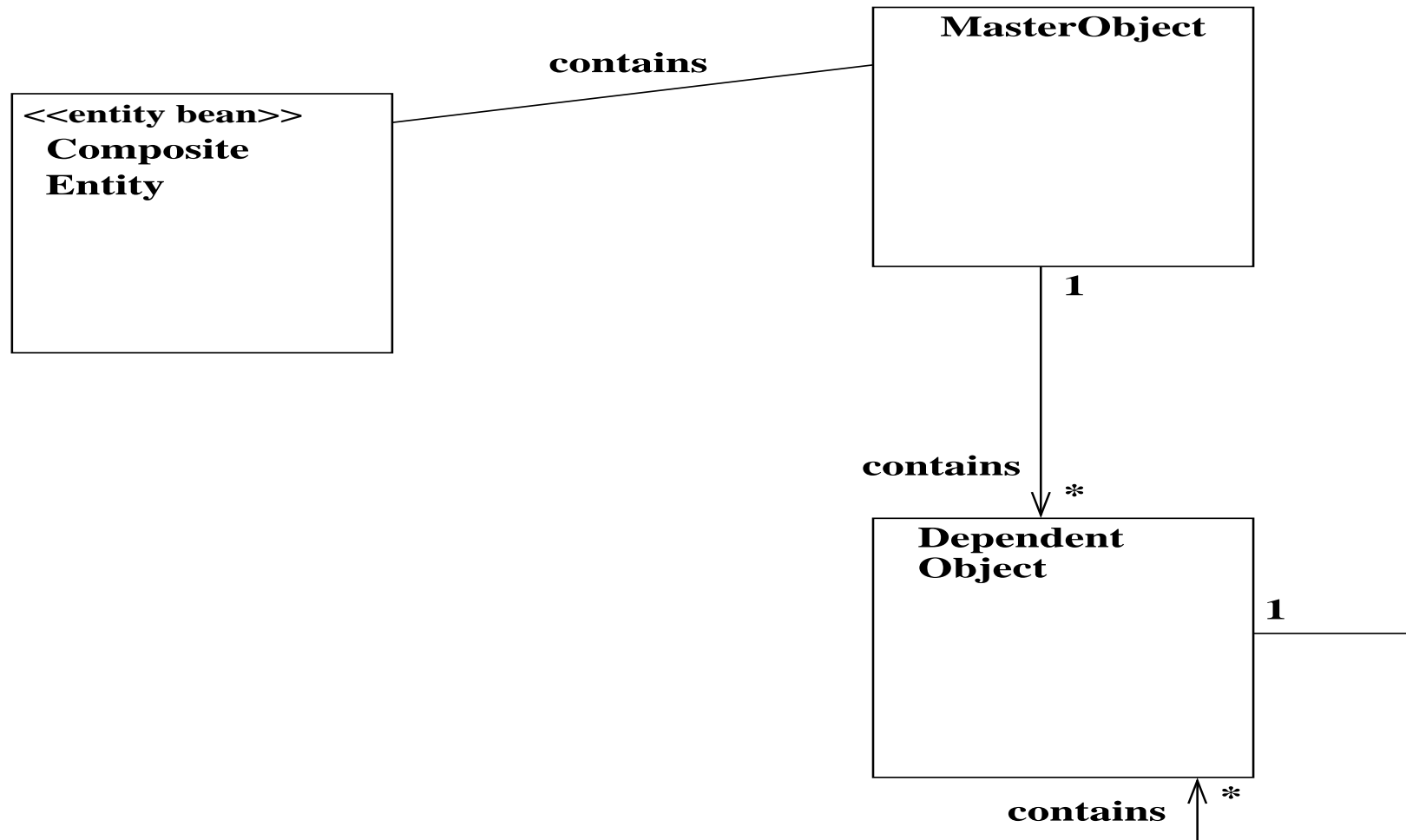


Composite Entity

This business tier pattern uses entity beans to manage a set of interrelated persistent objects, to improve efficiency.

If entity beans are used to represent individual persistent objects (eg, rows of a relational database table), this can cause inefficiency in access due to the potentially remote nature of all entity bean method calls. Also it leads to very many classes.

Instead, this pattern groups related objects into single entity beans.



Class diagram of composite entity

The elements of the pattern are:

- *Composite Entity*: coarse-grained entity bean. It may itself be the ‘master object’ of a group of entities, or hold a reference to this. All accesses to the master and its dependents go via this bean.
- *Master Object*: main object of a set of related objects, eg, a ‘Bill’ object has subordinate ‘Bill Item’ and ‘Payment’ objects.
- *Dependent Object*: subordinate objects of set. Each can have its own dependents. Dependent objects cannot be shared with other object sets.

Parts of a master object belong to the same composite entity set as the master.

An example could be:

```
public class BillEntity implements EntityBean
{ public int billTotal = 0;
  public List billItems = new ArrayList(); // of BillItem
  public List payments = new ArrayList(); // of Payment
  ...
}
```

Subordinate classes, *BillItem* and *Payment*, do not need their own entity beans. Can be standard Java classes.

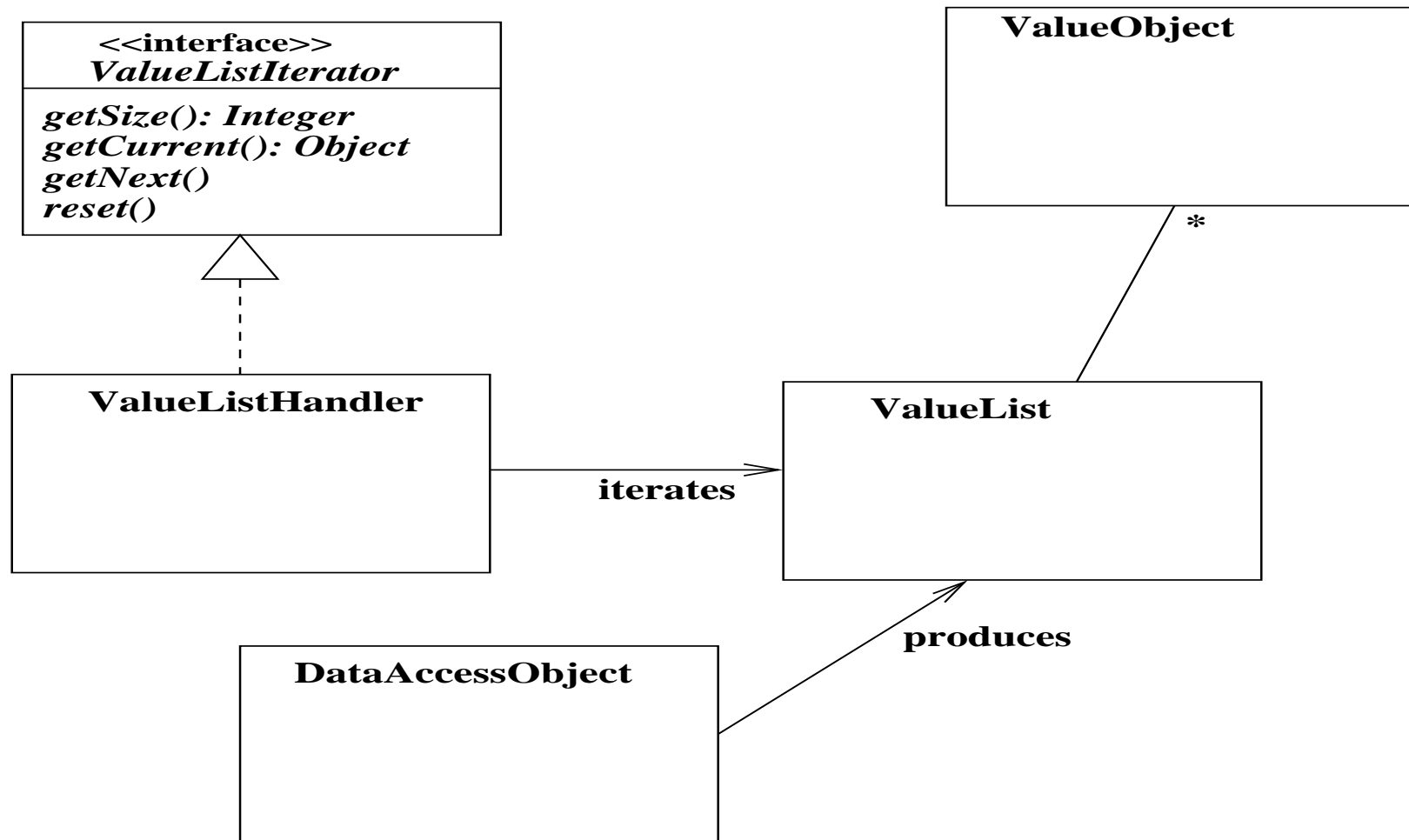
Guidelines for composite objects:

- If there is association $E \rightarrow D$ and no other association to D , put E and D in same entity bean.
- Put subclasses of a class in same entity bean as it.
- Put aggregate part classes of class in same entity bean as it.
- If D is a target of several associations $E \rightarrow D$, $F \rightarrow D$, etc., choose the association through which most accesses/use cases will be carried out, and make D part of the same entity bean as the class at the other end of that association.

Value List Handler

This integration tier pattern has the purpose to manage a list of data items/objects to be presented to clients. It provides an iterator-style interface allowing navigation of such lists.

The result data lists produced by database searches can be very large, so it is impractical to represent the whole set in memory at once. This pattern provides a means to access result lists element by element.



Structure of value list

The elements of this pattern are:

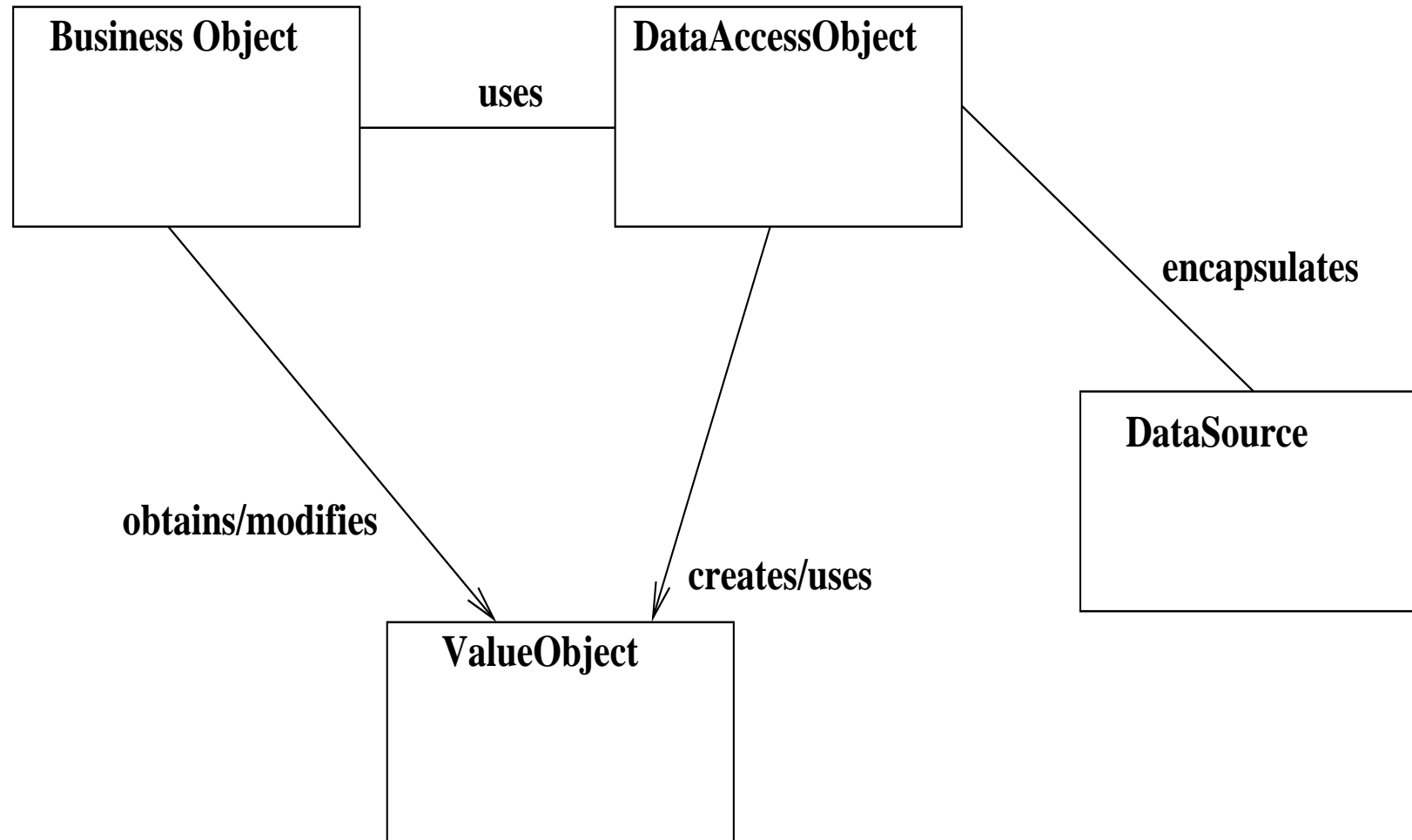
- *ValueListIterator*: an interface with operations such as *getCurrentElement()*, *getNextElements(int number)*, *resetIndex()* to navigate along the data list.
- *ValueListHandler*: implements *ValueListIterator*.
- *DataAccessObject*: implements the database/other data access.
- *ValueList*: the actual results of a query. Can be cached.

Data Access Object

This integration tier pattern abstracts from details of particular persistent data storage mechanisms, hiding these details from the business layer.

The variety of different APIs used for persistent data storage (JDBC, JSON, XML, B2B services, etc) makes it difficult to migrate a system if these operations are invoked directly from business objects.

This pattern decouples the business layer from specific data storage technologies, using the DAO to interact with a data source instead.



Structure of data access object

This pattern has the following elements:

- *Business Object*: requires access to some data source. It could be a session bean, entity bean, etc.
- *Data Access Object*: allows simplified access to the data source. Hides details of data source API from business objects.
- *Data Source*: actual data. Could be a relational or object-oriented database, or XML dataset, etc.
- *Value Object*: represents data transmitted as a group between the business and data access objects.

Factory Method or Abstract Factory patterns can be used to implement this pattern, to generate data access objects with the same interfaces, for different databases.

Summary

In this part we have described specification and design techniques for enterprise systems.

The key points are:

- Enterprise information systems typically involve distributed processing, and multiple client applications using same core business functionality and data.
- Business tier of an EIS can be structured around session beans and entity beans, which directly reflect high-level PIM specification of EIS as use cases and class diagrams.
- For each constraint of system there should be some component within business tier which is responsible for maintaining the constraint.

Class invariants and local business rules of a class can be maintained by entity bean which implements semantics of class.

Constraints which link states of two or more classes (and constraints on explicit associations between these classes) can either be maintained by an entity bean which encapsulates data of all these classes – in case that these classes represent closely related data, such as a main class and one or more subordinate auxiliary classes – or by a session bean which invokes operations of all entity beans implementing the classes.

- Design patterns for EIS can be defined to simplify EIS development and provide standard solutions to common EIS design problems.