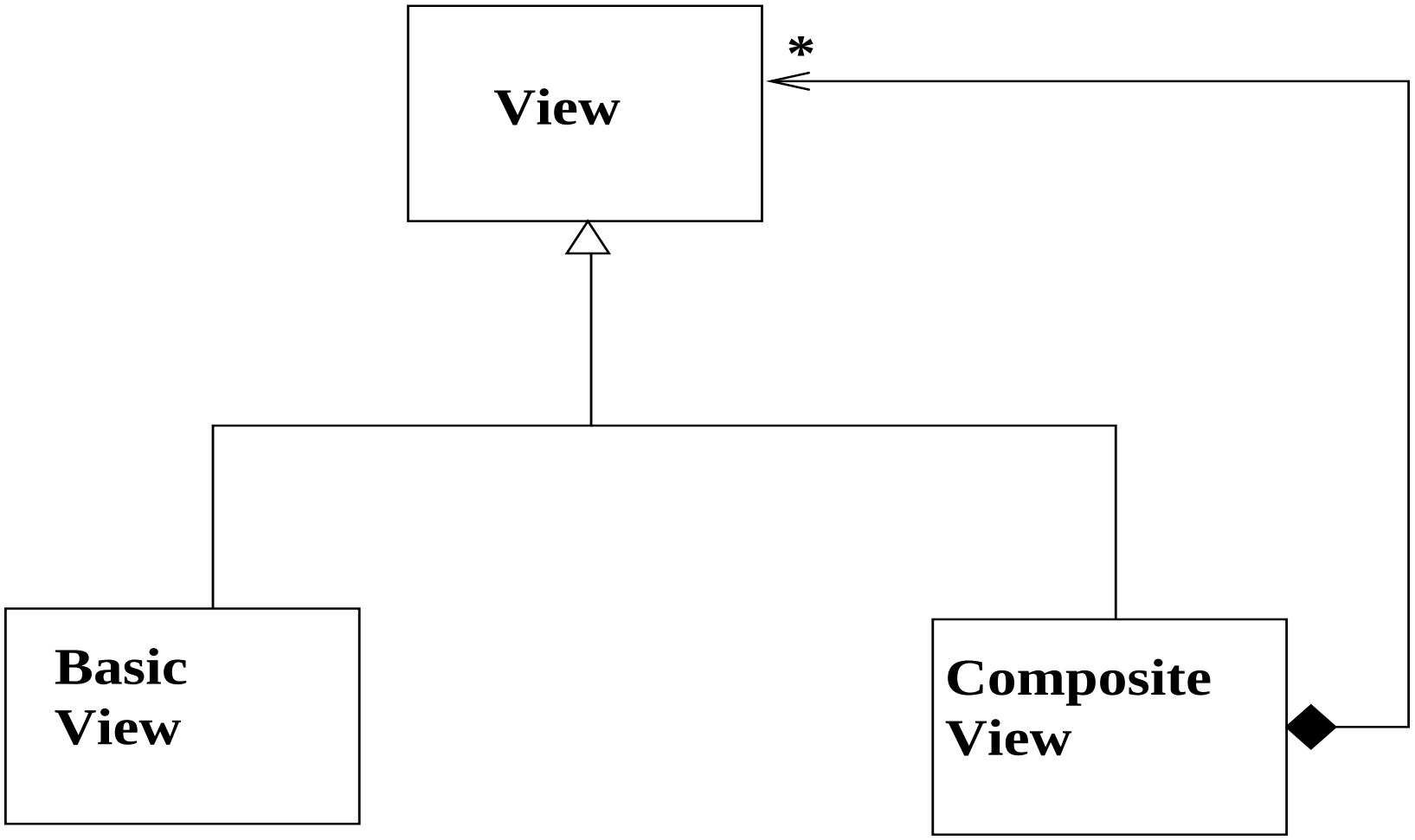
Composite View

This presentation tier pattern has the purpose of managing views which are composed from multiple subviews.

Complex web pages are often built out of multiple parts, eg, navigation section, news section, etc. Hard-coding page layout and content provides poor flexibility.

The pattern allows views to be flexibly composed as structures of objects.



Class diagram of composite view

The elements of the pattern are:

- *View*: a general view, either atomic or composite.
- *View Manager*: organises inclusions of parts of views into a composite view.
- *Composite View*: a view that is an aggregate of multiple views. Its parts can themselves be composite.

An example of this pattern in Java could be the `<jsp:include page = "subview.jsp">` tag in JSP, used to include subviews within a composite JSP page.

Other approaches include custom JSP tags and XSLT (if data is stored as XML).

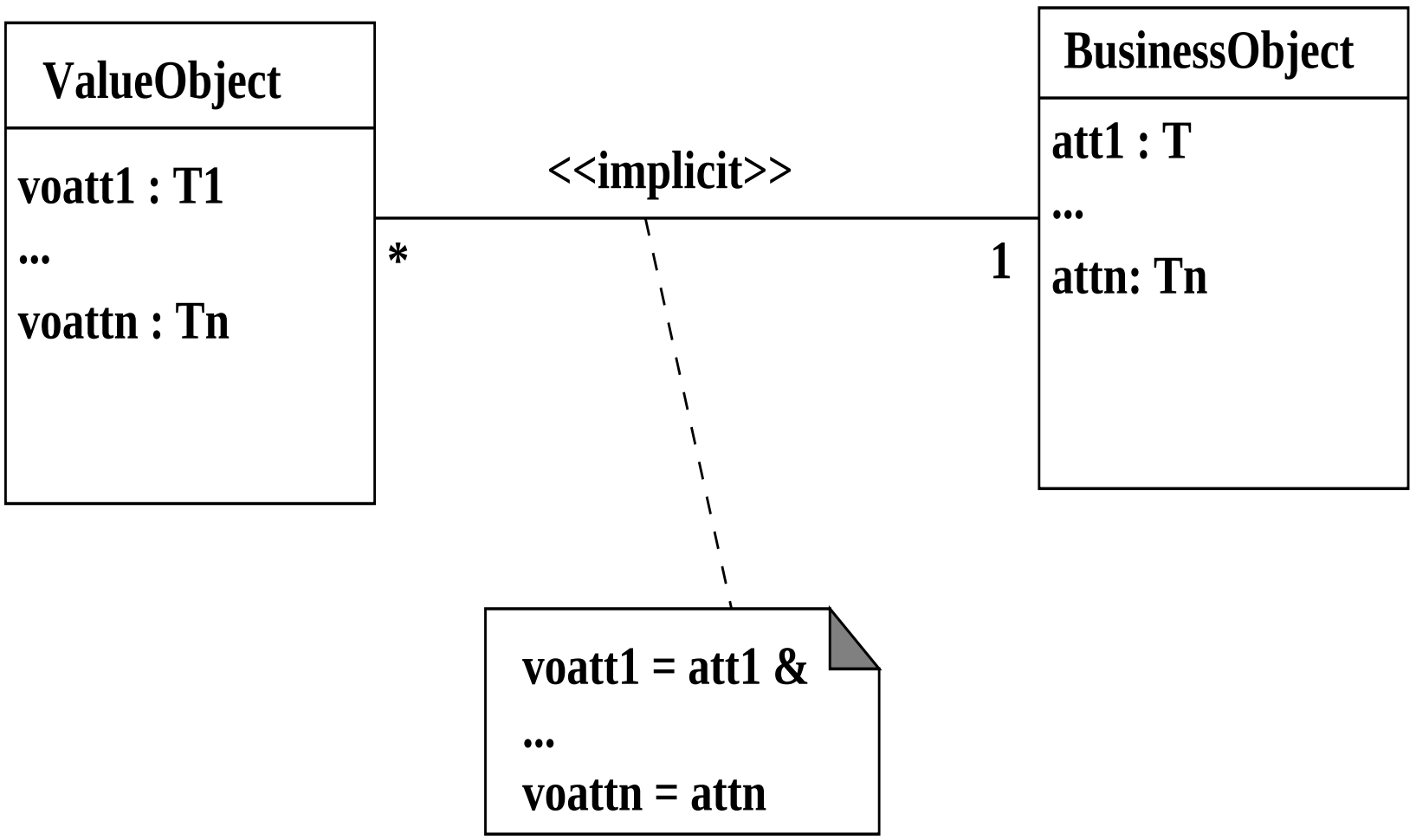
Value Object

This business tier pattern has the purpose to improve the efficiency of access to persistent data (eg, in entity beans) by grouping data and transferring data as a group of attribute values of each object.

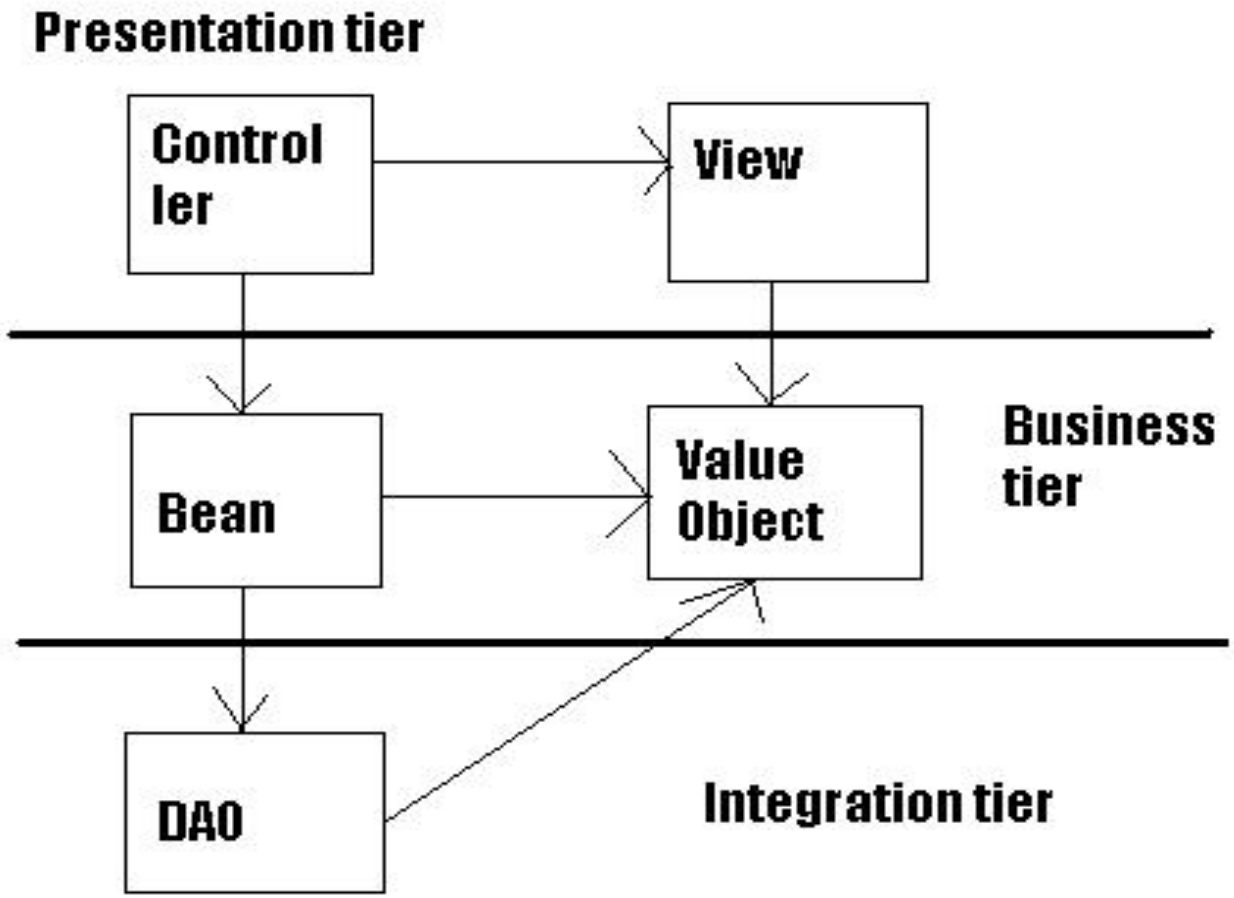
It is inefficient to get attribute values of a bean one-by-one by multiple *getatt*() calls, since these calls are potentially remote.

The pattern reduces data transfer cost by transferring data as packets of values of several attributes. Reduces number of parameters in bean operations.

Can transfer data between presentation and business tiers, and between integration and business tiers.



Class diagram of value object



Architecture diagram of value object

The elements of the pattern are:

- *Business Object*: can be a session or entity bean. Holds business data. It is responsible for creating and returning the value object to clients on request.
- *Value Object*: holds copy of values of attributes of business object. It has a constructor to initialise these. Its own attributes are normally public.

This pattern satisfies an invariant

$$voatt = att$$

for each attribute *att* of the business object and corresponding attribute *voatt* of the value object.

Some example code is:

```
public class BusinessObject implements EntityBean
{ private T1 att1;
```

```
  ...
  private Tn attn;

  ...

  public ValueObject getData()
  { return new ValueObject(att1,...,attn); }
}

public class ValueObject implements Serializable
{ public T1 voatt1;
  ...
  public Tn voattn;

  public ValueObject(T1 v1, ..., Tn vn)
  { voatt1 = v1;
    ...
    voattn = vn;
```

```
  }
}
```

The value object can also be used to update business objects via a *setData*(*ValueObject vo*) method.

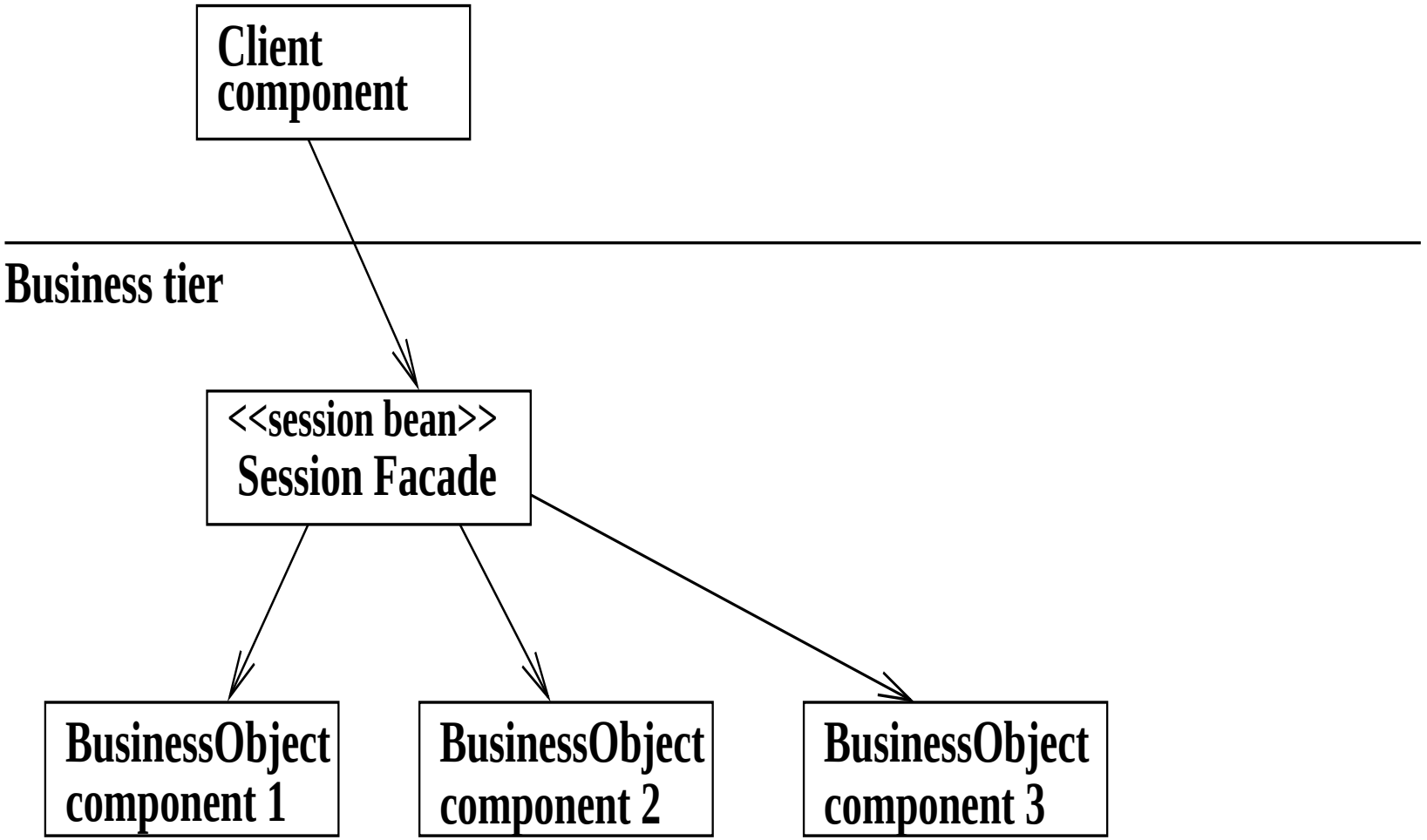
Example: *getDetails* method of *CustomerControllerBean*.

Session Facade

This business tier pattern aims to encapsulate the details of complex interactions between business objects. A session facade for a group of business objects manages these objects and provides a simplified coarse-grain set of operations to clients.

Interaction between a client and multiple business objects may become very complex, with code for many use cases written in the same class.

Instead this pattern groups related use cases together in session facades.



Architecture of session facade

The elements of the pattern are:

- *Client*: client of session facade, which needs access to the business service.
- *SessionFacade*: implemented as a session bean. It manages business objects and provides a simple interface for clients.
- *BusinessObject*: can be session beans or entity beans or data.

Several related use cases can be dealt with by a single session facade – if these use cases have mainly the same business objects in common.

Example: *CustomerControllerBean*, *AccountControllerBean*, *TxControllerBean*.