Pattern: Hierarchical Views

Abstract

Hierarchical Views is a pattern that describes a proven and common interface between an application kernel and the database access layer in a three-layer-architecture.

Example

Consider the detail of our order processing system shown in Figure 5. There may be use cases working with invoices having the structure depicted on the right side. Note that this invoice has a hierarchical structure with two levels of indirection. The use case may start with an order number and then navigate to the various items and their articles.

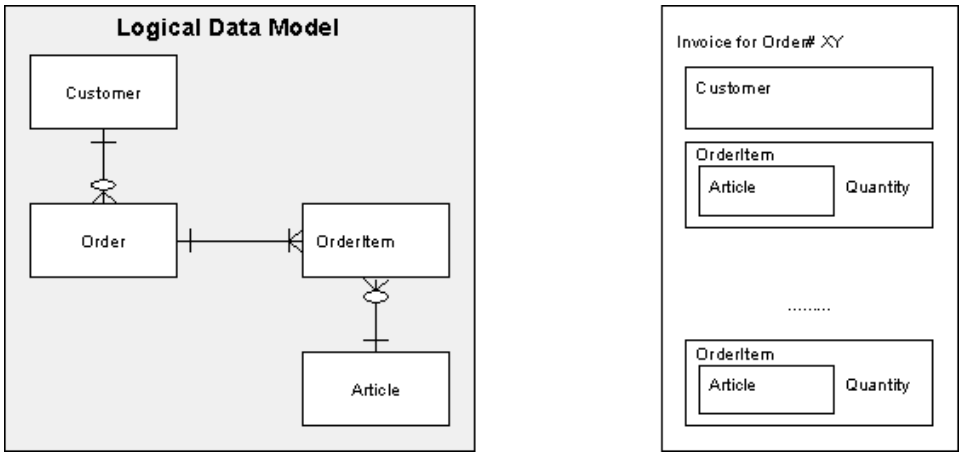


Figure 5 : A detail of our order processing system's logical data model. The left side shows the E/R diagram in third normal form, the right side the structure of an invoice, composed of these entities.

Context

You have decided to use the Relational Database Access Layer to decouple your physical database from the logical data model of the application kernel.

Problem

What interface should the database access layer present to the application kernel?

Forces

Besides the general considerations listed in the Introduction you might consider the following set of forces:

- *Complexity and power versus development cost:* The design of the interface is the central place where you can influence the power and complexity of your interface on the one hand. On the other hand this is also where you decide on the cost of implementing a powerful expensive interface.
- *Complexity versus ease-of-use:* The more complex and powerful you design the interface for applications accessing a database, the harder it will become to program it. If your goal is factoring out database concerns you should try to hide as many of them as possible. This will result in a simple to use but less powerful interface.
- *Mass problems:* A large data model contains hundred or more entities. Manually writing wrappers or embedded SQL-code for hundreds of entities is a boring and expensive task. Boring tasks are always error prone. A generic solution enables you to use macro expansion, generators or templates for database programming.

Solution

Express the interface in terms of the domain's problem space, that is as relational data model. Start at one point (or entity) of the data model and use foreign key relations to navigate to the other points of interest. Construct a directed acyclic graph (DAG) during navigation. Label every node with the entity, attributes of interest and selection predicates. Label every edge with the foreign key you have used for navigation and its cardinality (one to one or one to many).

Structure

Figure 6 shows a graph representation equivalent to the invoice of the left side.

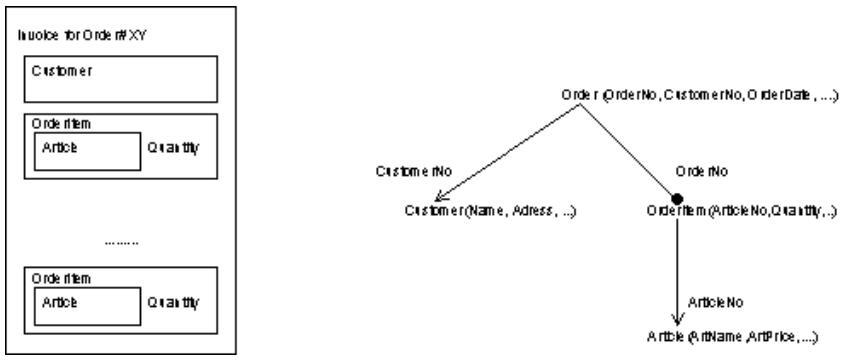
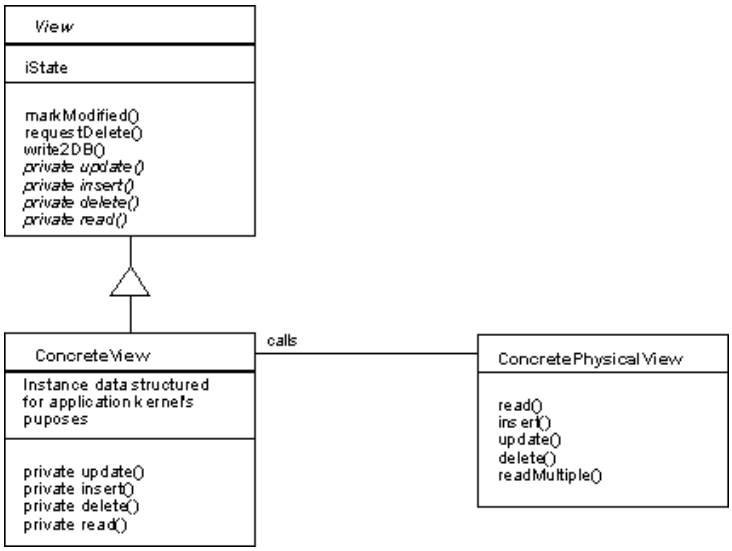


Figure 6 : A detail of our order processing system's logical data model. The left side shows the invoice known from Figure 5. The right side shows a DAG-like description of the data contained in the invoice. Every node or leaf of the DAG represents an entity of the logical data model.

To transform this graph into a *Hierarchical View*, write a ConcreteView derived from View[1]. The ConcreteView is the root of the DAG. Define a domain level class for any one of the nodes. Use aggregation to implement *to one* relationships in the graph. Use containers to implement *to many* edges of the graph. A ConcreteView constructed this way, fills its domain level attributes from ConcretePhysicalViews. The suitable ConcretePhysicalViews may be found using hard coded knowledge or the Query Broker.

The database access layer should be able, to treat all ConcreteViews uniformly. Therefore, View defines their common interface to the other classes of the access layer.



Example Resolved

Listing 1 shows the declarations of the invoice example, Listing 2 contains the code to process the invoice.

```
Struct Customer {
    CustomerKeyType     iCustNumber;
   ...  // other properties of the Customer in the logical data model
};
struct Article {
    ArticleNumberType   iArticleNumber;
   ...  // Other properties
};
struct OrderItem {
    Article             iArticle;
    QuantityType        iQuantity;
};
```

```
class OrderInvoiceView : public View {

public:

    OrderKeyType        iOrder;

    Customer            iCustomer;

    Vector<OrderItem>   iItems;  // Any other container will also do

    Money               iSumOfInvoice;

private:

    // private methods you need to obtain data and write data
    // to PhysicalViews



    virtual void update ( void );

    virtual void insert ( void );

    virtual void remove ( void );

    virtual void read ( void );

};
```

Listing 1 : The declarations for the invoice example.

The code of Listing 2 is free of database aspects and follows the logical data model. The denormalized physical data model is invisible from the application code. There are only two lines that deal with persistence: The ViewFactory::getView() command gets data from the access layer. The pos->markModified() method tags the SumOfInvoice to write itself back to the database.

```
Void Order::processInvoice (OrderKeyType anOrder) {

    // get the data from the database. We only specify the primary key

   // and leave the rest to the access layer

    OrderInvoiceView * pInvoice =
        (OrderInvoiceView *) ViewFactory::getView( anOrder );



    // process invoice items.

    ItemIterator itemIter = pInvoice->iItems.begin();

    for (; itemIter != iItems.end(); itemIter++) {

        itemIter->iSumOfInvoice +=

            ( itemIter->iQuantity *

          itemIter->iArticle.iArticlePrice );

    }



    // the view has been changed, so mark it

    pInvoice->markModified();

}
```

Listing 2 Implementation of processInvoice. The example demonstrates iteration through the items of an order and sums up the prices of all items in the iSumOfInvice property. Note that we traverse two levels of indirection in the logical data model. For reasons of simplicity we omitted the transaction brackets around Order::processInvoice as well as some obvious type definitions.

Consequences

Besides the general consequences described in the Introduction the consequences of a tree like interface are:

- *Inheritance and polymorphism*: The access layer contains no built-in precautions for handling inheritance or polymorphism. There is no thing like one view inheriting from another or a resolution of inheritance at a certain point of a view. This will work fine for most business information systems specified using soft object orientation (as practiced by Denert [Den91]). Such business information systems often use data encapsulation only but no inheritance or polymorphism. Anyway check whether this suits your problem domain.

- *Complexity of the interface:* The interface is minimal because it offers only the basic features the application kernel needs. However, you have to invest effort in generators or templates. There may be a full pay back during tuning but it may as well take several maintenance cycles before you reach break even. Once you have finished the generators, defining new ConcreteViews is a matter of minutes.

- *Interface style:* An application using Hierarchical Views follows the structure of the logical data model. The logical data model determines the structure of the code using it. Instead, with an object/relational access layer the object model follows the internal structure of the domain.

- *Ease of use and requirements of the application kernel:* Hierarchical Views reflect only the domain logic while supporting exactly the navigation that the corresponding use cases need. Calling the access layer is simple because the Hierarchical Views encapsulate database specific functions.

- *Decoupling:* Hierarchical Views completely decouple the application kernel from the physical data model. This enables you to tune the database any way you want to without affecting the application kernel's code. The resulting performance gain is much higher than the loss caused by the additional level of indirection the Hierarchical Views introduce.

Implementation

You may define the structure of the ConcreteViews using text files or a specialized tool [Würt96]. This allows automatic generation of the ConcreteViews for statically typed languages or even runtime definition for dynamically typed languages.

Variants

Many applications are a collection of mostly simple use cases. They need views with only a single level of indirection (like an entity and its dependent entity). In these cases the ConcretePhysical Views encapsulate the database access code and provide a sufficiently clean interface to the application, saving the Query Broker and the Hierarchical Views. However, this variant is not suitable for complex use cases that may touch a two digit number of entities in a single use case. As an example, consider insurance applications.

A more complex variant allows retrieval of historic data. You need this variant if you are not interested only in the current state of a contract but in its state at a given time [Sch96]. To navigate the data model, you have to enrich conditions and navigation edges with expressions for time based navigation [Würt96]. Insurance companies often need these features.

Related Patterns

You may use a Query Broker to decouple Hierarchical Views from the underlying Physical Views.

Use a View Cache to avoid multiple database accesses for the same physical data.

Known Uses

VAA, a standard architecture for German insurance companies, uses this pattern with time navigation [VAA95]. The corresponding *Data Manager Component* is currently under construction. Württembergische Versicherung [Würt96] develops a *Data Manager* using Hierarchical Views and a tool to define them.

Many of sd&m's projects have used the simple variant (1:n views) of the Hierarchical Views pattern. These projects use scripting languages to automatically generate views from view descriptions [Den91].

---

[1] You may model this approach in 3GL using structures instead of domain level classes. We shall return to this issue in the implementation section.