The Dilemma of Encapsulation vs. Query Optimization

Michael Blaha (blaha@computer.org)
Modelsoft Consulting Corp
www.modelsoftcorp.com

Abstract

There is a fundamental, irreconcilable conflict between the goals of encapsulation and the goals of query optimization [1] [2].

The object-oriented literature emphasizes encapsulation. An object should access only objects that are directly related. An object can access indirectly related objects via the methods of intervening objects. Encapsulation increases the resilience of an application; local changes to an application model cause local changes to application code.

On the other hand, database management system (DBMS) optimizers take a logical request and generate an execution plan. If queries are broadly stated, the optimizer has greater freedom for devising an efficient plan.

Thus encapsulation boosts resilience but limits optimization potential. On the other hand, broadly stating queries is conducive to optimization, but a small change to an application can affect many queries.

1. Encapsulation

The Law of Demeter [3] eloquently states the principle of encapsulation. "For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:

- The argument classes of M (including C).
- The instance variable classes of C."

Restating the Law in UML terms, an object should access only objects that are directly related (the objects' classes are directly connected by an association).

The intent is that a method should have limited knowledge of a class model. A method must be able to traverse links to obtain its neighbor objects and must be able to call methods on them, but it should not traverse a second link from a neighbor to objects of a third class because the second link is not directly visible to it. Instead call a method on the neighbor object to traverse nonconnected objects. Then if the association network changes, there is reduced disruption to methods.

Similarly, avoid applying a second method to the result of a method call unless the class of the result is already known as an attribute, argument, or neighbor, or the result class is from a lower-level library. Instead, write a new method on the original target class to perform the combined method.

There are two primary motivations for encapsulation:

- **Simpler modifications**. Encapsulation minimizes a method's dependencies on other classes.
- **Less programming complexity**. It restricts the number of types the programmer must be aware of when writing a method.

Consider the alternative without encapsulation. Taken to an extreme, a method could traverse all associations of the class model.

2. Query Optimization

Query optimization has a different purpose—freedom for the DBMS to optimize. Many DBMSs—especially relational DBMSs (RDBMSs)—have sophisticated optimizers and may use efficient algorithms that an application programmer may overlook. In addition, many DBMSs can revise their choice of algorithms as tuning structures and data distribution change over time.

RDBMS performance will usually be best if you join multiple tables together in a single SQL statement, rather than disperse the tables across multiple statements. Note the contradiction to encapsulation! Encapsulation encourages developers to minimize combinations of tables in queries; query optimization encourages exactly the opposite—large combinations of tables.

This conflict might seem to be between OO languages and RDBMSs. However, it is really between OO languages and DBMSs (both relational and OO). The conflict is prominent for RDBMSs because they emphasize nonprocedurality with the SQL language. However, as OO databases become more advanced, their query optimizers will improve and raise the same dilemma.

3. An Example

Figure 1 shows a portion of a UML class model for a flight reservation system. *Airlines* fly between *Airports*. A *FlightDescription* refers to the published description of air travel between two *Airports*. In contrast, a *Flight* refers to the actual travel made by an airplane on a particular date.

The *frequency* indicates the days of the week for which the *FlightDescription* applies. The *startEffectiveDate* and *stopEffectiveDate* indicate the time period for which the *FlightDescription* is in effect.

A sample *FlightDescription* is TW 250 from St. Louis to Milwaukee. In the fall of 1995, TW 250 was scheduled to depart at 7:42 AM, had a scheduled duration of one hour eighteen minutes, was scheduled to be flown by a DC9 plane, and operated every day of the week except Sunday. A sample *Flight* is TW 250 on October 23, 1995, which actually departed at about 7:45 AM and had an actual duration of one hour five minutes.

Each *FlightDescription* involves an *AircraftDescription*. For example, the TW 250 *FlightDescription* was intended to be flown with a DC9. In contrast, a *Flight* involves an actual physical *Aircraft* having a unique *registrationNumber*.

The model permits us to answer queries such as the following: Find the *Airline* that flies an *Aircraft.* Encapsulation would lead to the following sequence of method calls. (Note that the only meaningful traversal is from *Aircraft.Flight.FlightDescription.Airline*.)

- *Aircraft.getAirline()* calls *Flight.getAirline()* for each of the *Flights* for an *Aircraft*. The class model permits an *Aircraft* to have multiple *Airlines*, though that is unintended. (The class model has no constraint to prevent it.) Code would have to check for this possibility and return an error if it is found.
- *Flight.getAirline()* calls *FlightDescription.getAirline()*.
- *FlightDescription.getAirline()* returns the *Airline*.

Alternatively we could write the following SQL query. (The code is for MS SQL Server and assumes a straightforward relational database design—the subject of a future article.)

```
SELECT DISTINCT(FD.airlineID)
FROM Aircraft AS Ac
INNER JOIN Flight AS F
    ON Ac.aircraftID = F.aircraftID
INNER JOIN FlightDescription AS FD
    ON F.flightDescriptionID = FD.flightDescriptionID
WHERE Ac.aircraftID = theGivenAircraft
```

Now consider the effect of a revision to the model. The model in Figure 1 is lacking in that it does not support through flights. Airlines routinely schedule flights that start in one city, stop in one or more intermediate cities, and terminate in a final city. The entire sequence of flights has a single flight number, yet has more than just origin and destination airports. Figure 2 adds *FlightLegDescription* to support through flights.

This revision affects the association between *Flight* and *FlightDescription*. Accordingly, *Flight.getAirline()* must be rewritten to call the new method *FlightLegDescription.getAirline()*. There are no other changes to our previous methods. So, with encapsulation, a small revision to the class model affects only the methods that are immediately involved. In contrast complex DBMS queries touch many classes and are more likely to require rewriting.

4. Analysis vs. Design

We should clarify one aspect of our discussion. The tradeoff of encapsulation vs. query optimization is a design issue

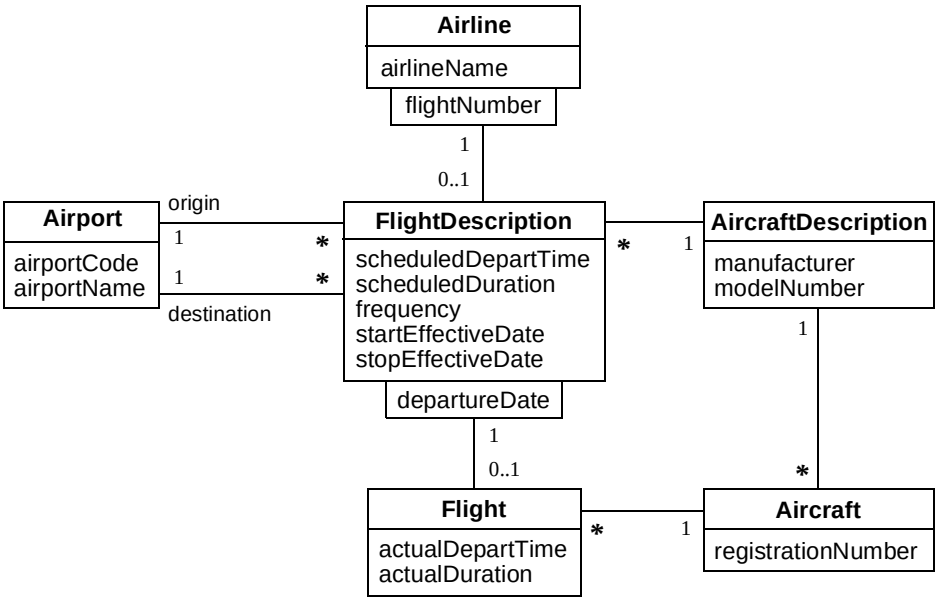

Figure 1 UML class model for an airline flight reservation system

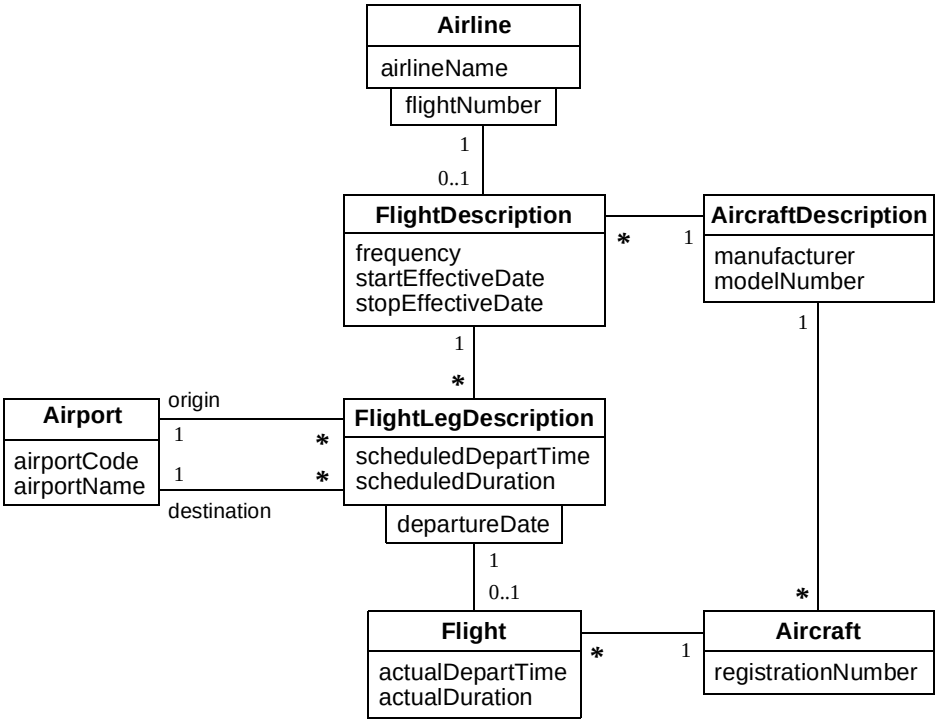

Figure 2 Revised UML class model for an airline flight reservation system adding *FlightLegDescription*

and is not about analysis. During analysis, the goal is to understand the application. The exact construction of the ultimate code is not a concern.

In contrast, design follows analysis and determines how to write the code to deliver the desired features and functionality. The principle of encapsulation comes into play during design as we are trying to make methods robust so that they are resilient to changes. Query optimization is also important during design because another goal is good performance.

5. What Should You Do About It?

We have explained the conflict between encapsulation and query optimization, but, as a developer, what can you do about it?

- **Programming applications**. There is no problem because programming languages do not have a query optimizer. Just encapsulate your code as the OO literature advises.
- **OO-DBMS applications**. You should usually tilt toward encapsulation. Currently, most OO-DBMSs have weak query optimizers, so there is little incentive for broadly stating queries.
- **RDBMS applications**. There is no simple answer. There are three different cases.
  - -- **Complex programming**. Encapsulate your code if the programming is intricate and performance degradation is not too severe.
  - -- **Easy programming and good query performance**. Broadly state queries if doing so improves RDBMS performance and the programming code and queries are relatively easy to write—and rewrite if the class model changes.
  - -- **Easy programming and poor query performance**. Somewhat paradoxically, you can sometimes improve performance by fragmenting queries. Query optimizers are imperfect, and occasionally you will need to guide the optimizer manually. In this case, you blend encapsulation and query optimization, writing queries that traverse multiple classes, but less than in the previous case.

6. References

[1] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Upper Saddle River, New Jersey: Prentice Hall, 1998.

[2] Michael Blaha and James Rumbaugh. *Object-Oriented Modeling and Design with UML, Second Edition*. Upper Saddle River, New Jersey: Prentice Hall, 2005.

[3] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. *OOPSLA'88* as *ACM SIGPLAN 23*, 11 (November 1988), 323–334.