Split Class Pattern

Derek Laufenberg
Director of Product Management
Versant Corporation, dlaufen@versant.com

Abstract

Sometimes a class's natural structure isn't the optimal design when database considerations are taken into account. The object level granularity of the object oriented databases cause more I/O than necessary when a class has a mixture of static and frequently updated attributes. Additionally, large data attributes, such as embedded blobs or arrays, might not be used with every transaction. Splitting the class into a few supporting classes can improve system performance without unduly complicating the application design.

Problem Detail

When an OO database client loads or stores an object, the entire object is transferred to or from the server. This is required by languages like C++ where the object must be fully loaded before it can be handed off to the rest of the application. This wastes bandwidth when there large portions of the class's attributes won't be used within the transaction. During updates, the unchanged attributes are also sent back to the database when committed, this too can waste server and network resources.

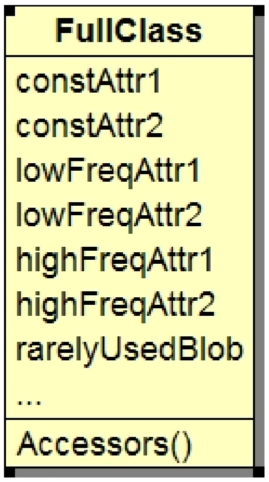

Figure 1 – Example class

Consider the class shown in Figure 1. In there, there are attributes that span the range from frequently changed to effectively constant after instance creation. There is also a blob that isn't always used with every transaction. Despite the logical grouping of attributes by the class, the different access patterns can be exploited to improve the application's database use.

Pattern Solution

Splitting the single class into multiple implementation classes is done by grouping attributes in supporting classes based on the application's use saves system bandwidth and improves performance. For example, rather than embedding the blob, it is moved to its own class. A reference is added in the main class to reach the blob and this reference is used to load the blob only when it's needed. This results a huge bandwidth savings for transaction that don't need the blob. This technique can be used with any large rarely used data member just as easily.

The attributes that change frequently are also pushed into another supporting class so their frequent updates don't require resending the blob or the other infrequently changed attributes. This split also limits the amount of data that gets locked during an update; this helps improve system concurrency.

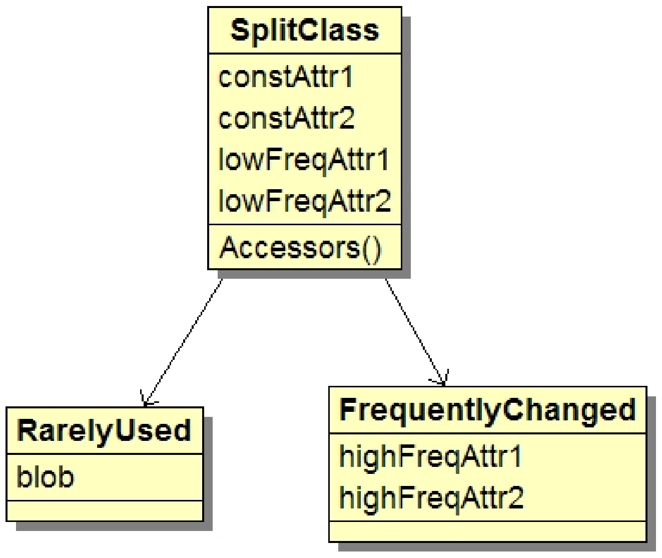

Figure 2 – Class split into multiple classes by application access patterns.

The original class or interfaced should act as a delegate for any operations on the support class objects. This keeps the logical cohesion offered by the original design.

Benefits

Good partitioning will greatly reduced network and disk IO when the objects are updated or when they avoid being loaded. Interfaces and delegation should be used to hide the splitting so as not to impact the overall design.

Drawbacks

The major draw back to this pattern is the added complexity of splitting the class into multiple parts.

For small classes the additional OID overhead might not justify the split.