

Why Object-Oriented Business Transaction Processing Systems Do Not Work ...as Well as We Would Like

Michael P. Anton
anton@bernstein.com

Problems often arise in implementing Object-Oriented (OO) solutions to business transaction processing systems. These problems can typically be classified into two types: *performance* and *scalability*. Often they manifest themselves (or become most apparent) when replacing legacy applications which were implemented with distinctly non-OO strategies. Transient objects are not usually troublesome; invariably the problems concern retrieval of objects from persistent storage (perhaps a relational database).

OO systems have an undisputed advantage in program correctness, flexibility, resiliency to changing requirements, and reuse. But arguments against their use for business transaction processing can be frequent and difficult to refute. The concerns usually involve performance, or the inability of OO systems to scale up. Faster hardware with more memory and distributed processing can quickly silence these arguments, but it seems that there is nothing inherent in OO systems causing these problems. On the other hand, perhaps there *are* some theoretical limitations to “pure” OO which leave hybrid solutions the only reasonable compromise.

Performance: They need to be fast. Objects provide an ideal mechanism to encapsulate business logic. However, instantiating an object can be an expensive process. Business transaction processing systems very often need to apply a single business rule or calculation for a very large number of objects. If an object has been generalized for use in many systems, it may have dozens or even hundreds of attributes. Retrieval of the object from persistent storage may take much more time than the time spent calling the method. Special care must be taken to eliminate retrieval of unnecessary attributes from persistent storage.

One approach¹ to solving this problem is to have *specialized constructors* optimized for different applications. If a system does not require certain attributes, it can call a constructor which is specialized for this (type of) application and only instantiates attributes that are likely to be used. However, this violates the encapsulation of the object and exposes its implementation to the system using it.

Another strategy that may ease the problem is *lazy instantiation*. Upon construction, the object’s attributes are not retrieved from persistent storage. Instead, the first time the attributes are needed, they are retrieved. This can avoid the overhead of retrieving unnecessary attributes, but may increase the number of calls that must be made to persistent storage.

¹ I avoid using the term *pattern* since these concepts need further development before this would be appropriate.

A better approach is *intelligent instantiation* (or “*live-and-learn*” instantiation). Since business transaction processing systems are operating in a similar manner over large numbers of objects, the class can keep track of which attributes were used in previous instantiations. When a new object is instantiated, the class dynamically modifies the attributes retrieved from persistent storage based on which attributes were used in other object instantiations. It may be desirable to store the state of the class itself between system executions. In this case, the class would learn from previous executions what attributes should be retrieved. Even if the objects are all retrieved from persistent storage before accessing any of an object’s methods, the class will still learn from previous executions and performance should improve.

Scaleable: They need to handle large volumes. Object-Oriented systems have a way of taxing hardware and operating systems in amazing ways. With a few lines of code one may easily instantiate hundreds of objects. And the most straightforward way to program this is often to create all the objects and then set the system in motion.

This, unfortunately, will not work well for business transaction processing systems. The system may have hundreds of thousands or even hundreds of millions of transactions. The objects may not fit in memory. Even if they could, one typically would not want to wait for all of the objects to be retrieved from persistent storage before the processing begins.

A much better model is to instantiate and process the objects in smaller batches.² This may be programmed explicitly in the system, but it is preferable to have this handled behind the scenes by an object cache or *middleware*. The business transaction processing system may transparently access objects in the cache without explicitly retrieving them from persistent storage.

When scaling up processing a thousand-fold, another concern is *recovery*. In the event of system failure during processing, it is essential to resume activity with the last completed transaction and to roll-back any incompletely processed transactions. Systems with fewer transactions may have the luxury of being able to roll-back the entire process and restart from the beginning. This is not feasible for typical business transaction processing systems where processing may take hours or even days.

Michael P. Anton is Manager of Fixed Income Systems at Sanford C. Bernstein & Co., Inc., an Investment Management and Research Firm in New York City. He holds a Master of Science in Engineering (Computer & Information Science) from the University of Pennsylvania and a Bachelor of Arts in Economics and Chemistry from Williams College in Massachusetts.

Mr. Anton has over eighteen years experience designing, building, and managing software projects. He has been involved in Object-Oriented Development for about nine years and since 1993 has contributed to dozens of projects in portfolio management, trading, and transaction processing.

² Where “smaller” may mean anything from one to thousands depending on the system.