# Difference between DAO and DTO

http://www.jguru.com/forums/view.jsp?EID=1344828

Data Access Object(DAO) and Data Transfer Objects(DTO) are distinct J2EE design patterns.

A DAO can be thought of as a proxy used by the business tier who is its client to retrieve values from any given resource such as a RDMS, LDAP server, XML repository, OODB, flat files and so forth. The public DAO interface stays the same even if the resource implementation mechanism changes.

The DTO a.k.a. Value Transfer Object(VTO), Value Object(VO) (Martin Fowler defines VO pattern differently) is used to expose several values in a bean like fashion. This provides a light-weight mechanism to transfer values over a network or between application tiers. It's regarded as a more coarse grain object and eliminates trivial network chatter that occurs when retrieving single values one at a time.

So a DAO certainly makes use of DTOs when servicing it's clients. However, DTOs are also of value when communicating with other tiers within your application design (not just the persistence tier).

DTOs may be implemented with varying degrees of mutability. Using a constructor to pass in all the values with only getters gives essentially a read only object. The client then may map the values pursuant to its business objectives or pass it straight through to the presentation layer for final disposition.

There's no reason to be dogmatic about the mutability of DTOs; it's an application by application consideration and as long as you are aware of the issues (such as aliasing) you should be free of guilt (but perhaps not ridicule!) by allowing mutators in your VOs when useful.

http://www.hardik4u.com/2012/08/what-is-difference-between-dto-and-dao.html

DAO is a class that usually has the CRUD operations like save, update, delete. DTO is just an object that holds data. It is JavaBean with instance variables and setter and getters.

A DAO can be thought of as a proxy used by the business tier who is its client to retrieve values from any given resource such as a RDMS, LDAP server, XML repository, OODB, flat files and so forth. Here is how it looks:

your code -> DAO -> JDBC

The DTO, Value Object(VO) is used to expose several values in a bean like fashion. This provides a light-weight mechanism to transfer values over a network or between different application tiers.

DTO will be passed as value object to DAO layer and DAO layer will use this object to persist data using its CRUD operation methods.

The advantage of the DAO layer is that if you need to change the underlying persistence mechanism you only have to change the DAO layer, and not all the places in the domain logic where the DAO layer is used from. The DAO layer usually consists of a smaller set of classes, than the number of domain logic classes that uses it. Should you need to change what happens behind the scene in the DAO layer, the operation is somewhat smaller, since it only affects the DAO layer.

The advantage of DTO layer is that it adds a good deal of flexibility to the service layer and subsequently to the design of the entire application. For example, if DTOs are used, a change in the requirements that forces a move to a different amount of data doesn't have any impact on the service layer or even the domain. You modify the DTO class involved by adding a new property, but leave the overall interface of the service layer intact.

http://stackoverflow.com/questions/14366001/dto-and-dao-concepts-and-mvc

DTO is an abbreviation for Data Transfer Object, so it is used to transfer the data between classes and modules of your application. DTO should only contain private fields for your data, getters, setters and constructors. It is not recommented to add business logic methods to such classes, but it is ok to add some util methods.

DAO is an abbreviation for Data Access Object, so it should incapsulate the logic for retrieving, saving and updating data in your data storage (a database, a filesystem, whatever). Here is an example how the DAO and DTO interfaces would look like:

interface PersonDTO {

String getName();

void setName(String name);

//.....

}

interface PersonDAO {

PersonDTO findById(long id);

void save(PersonDTO person);

//.....

}

The MVC is a wider pattern. The DTO/DAO would be your model in the MVC pattern. It tells you how to organize the whole application, not just the part responcible for data retrieval.

As for the second question, if you have a small application it is comletely OK, however if you want to follow the MVC pattern it would be better to have a separate controller, which would contain the business logic for your frame in a separate class and dispatch messages to this controller from the event handlers. This would separate your business logic from the view.

# Idempotent

http://www.allapplabs.com/glossary/idempotent.htm

If methods are written in such a way that repeated calls to the same method do not cause duplicate updates, the method is said to be "idempotent."

In mathematics an idempotent element, or an idempotent for short, is anything that, when multiplied by itself, gives itself as result. For example, the only two real numbers which are idempotent are 0 and 1.

In user interface design, a button can be called "idempotent" if pressing it more than once will have the same effect as pressing it once. For example, a "Pause" button is not idempotent if it toggles the paused state. On the other hand, if pressing it multiple times keeps the system paused and pressing "Play" resumes, then "Pause" is idempotent. This is useful in interfaces such as infrared remote controls and touch screens where the user may not be sure of having pressed the button successfully and may press it again. Elevator call buttons are also idempotent, though many people think they are not.

Idempotent means no matter how many times you call the operation the result will be the same

http://stackoverflow.com/questions/1077412/what-is-an-idempotent-operation

An idempotent operation can be repeated an arbitrary number of times and the result will be the same as if it had been done only once. In arithmetic, adding zero to a number is idempotent.

Idempotence is talked about a lot in the context of "RESTful" web services. REST seeks to maximally leverage HTTP to give programs access to web content, and is usually set in contrast to SOAP-based web services, which just tunnel remote procedure call style services inside HTTP requests and responses.

REST organizes a web application into "resources" (like a Twitter user, or a Flickr image) and then uses the HTTP verbs of POST, PUT, GET, and DELETE to create, update, read, and delete those resources.

Idempotence plays an important role in REST. If you GET a representation of a REST resource (eg, GET a jpeg image from Flickr), and the operation fails, you can just repeat the GET again and again until the operation succeeds. To the web service, it doesn't matter how many times the image is gotten. Likewise, if you use a RESTful web service to update your Twitter account information, you can PUT the new information as many times as it takes in order to get confirmation from the web service. PUT-ing it a thousand times is the same as PUT-ing it once. Similarly DELETE-ing a REST resource a thousand times is the same as deleting it once. Idempotence thus makes it a lot easier to construct a web service that's resilient to communication errors.

**ACID**

http://en.wikipedia.org/wiki/ACID

In computer science, ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction. The chosen initials refer to the acid test.

**Atomicity**

Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

**Consistency**

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors do not violate any defined rules.

**Isolation**

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.[citation needed]

**Durability**

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

Examples

The following examples further illustrate the ACID properties. In these examples, the database table has two columns, A and B. An integrity constraint requires that the value in A and the value in B must sum to 100. The following SQL code creates a table as described above:

CREATE TABLE acidtest (A INTEGER, B INTEGER CHECK (A + B = 100));

**Atomicity failure**

Assume that a transaction attempts to subtract 10 from A and add 10 to B. This is a valid transaction, since the data continue to satisfy the constraint after it has executed. However, assume that after removing 10 from A, the transaction is unable to modify B. If the database retained A's new value, atomicity and the constraint would both be violated. Atomicity requires that both parts of this transaction, or neither, be complete.

**Consistency failure**

Consistency is a very general term which demands that the data must meet all validation rules. In the previous example, the validation is a requirement that A + B = 100. Also, it may be inferred that both A and B must be integers. A valid range for A and B may also be inferred. All validation rules must be checked to ensure consistency.

Assume that a transaction attempts to subtract 10 from A without altering B. Because consistency is checked after each transaction, it is known that A + B = 100 before the transaction begins. If the transaction removes 10 from A successfully, atomicity will be achieved. However, a validation check will show that A + B = 90, which is inconsistent with the rules of the database. The entire transaction must be cancelled and the affected rows rolled back to their pre-transaction state. If there had been other constraints, triggers, or cascades, every single change operation would have been checked in the same way as above before the transaction was committed.

**Isolation failure**

To demonstrate isolation, we assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation.

Consider two transactions. T1 transfers 10 from A to B. T2 transfers 10 from B to A. Combined, there are four actions:

T1 subtracts 10 from A.

T1 adds 10 to B.

T2 subtracts 10 from B.

T2 adds 10 to A.

If these operations are performed in order, isolation is maintained, although T2 must wait. Consider what happens if T1 fails half-way through. The database eliminates T1's effects, and T2 sees only valid data.

By interleaving the transactions, the actual order of actions might be:

T1 subtracts 10 from A.

T2 subtracts 10 from B.

T2 adds 10 to A.

T1 adds 10 to B.

Again, consider what happens if T1 fails halfway through. By the time T1 fails, T2 has already modified A; it cannot be restored to the value it had before T1 without leaving an invalid database. This is known as a write-write failure,[citation needed] because two transactions attempted to write to the same data field. In a typical system, the problem would be resolved by reverting to the last known good state, canceling the failed transaction T1, and restarting the interrupted transaction T2 from the good state.

**Durability failure**

Assume that a transaction transfers 10 from A to B. It removes 10 from A. It then adds 10 to B. At this point, a "success" message is sent to the user. However, the changes are still queued in the disk buffer waiting to be committed to the disk. Power fails and the changes are lost. The user assumes (understandably) that the changes have been made.

**Implementation**

Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons. For instance, the system may have no room left on its disk drives, or it may have used up its allocated CPU time.

There are two popular families of techniques: write ahead logging and shadow paging. In both cases, locks must be acquired on all information that is updated, and depending on the level of isolation, possibly on all data that is read as well. In write ahead logging, atomicity is guaranteed by copying the original (unchanged) data to a log before changing the database.[dubious – discuss] That allows the database to return to a consistent state in the event of a crash.

In shadowing, updates are applied to a partial copy of the database, and the new copy is activated when the transaction commits.

**Locking vs multiversioning**

Many databases rely upon locking to provide ACID capabilities. Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify it until the first transaction succeeds or fails. The lock must always be acquired before processing data, including data that are read but not modified. Non-trivial transactions typically require a large number of locks, resulting in substantial overhead as well as blocking other transactions. For example, if user A is running a transaction that has to read a row of data that user B wants to modify, user B must wait until user A's transaction completes. Two phase locking is often applied to guarantee full isolation.[citation needed]

An alternative to locking is multiversion concurrency control, in which the database provides each reading transaction the prior, unmodified version of data that is being modified by another active transaction. This allows readers to operate without acquiring locks, i.e. writing transactions do not block reading transactions, and readers do not block writers. Going back to the example, when user A's transaction requests data that user B is modifying, the database provides A with the version of that data that existed when user B started his transaction. User A gets a consistent view of the database even if other users are changing data. One implementation, namely snapshot isolation, relaxes the isolation property.

Distributed transactions[edit]

Guaranteeing ACID properties in a distributed transaction across a distributed database where no single node is responsible for all data affecting a transaction presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes, because of a failure on another node. The two-phase commit protocol (not to be confused with two-phase locking) provides atomicity for distributed transactions to ensure that each participant in the transaction agrees on whether the transaction should be committed or not.[citation needed] Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants) and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transaction.