The Persistence Manager

Stephen McHenry

A discussion of implementation issues for the Persistence Manager of Distributed Object Systems.

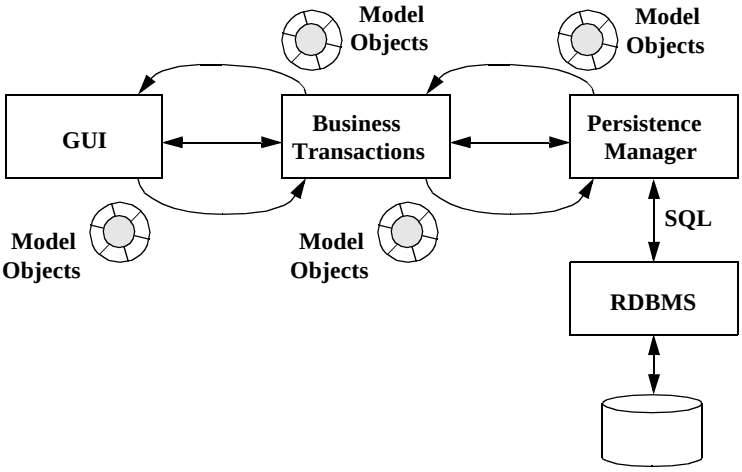
This document discusses implementation issues facing the Persistence Manager, which is used to save and restore objects from the database.

Introduction to the Persistence Manager

The role of the persistence manager in distributed object systems is to be an interface between the database and the rest of the system. It always lives on the same node as the RDBMS system itself, to eliminate the need to issue SQL across the network using the network transport mechanism of the database vendor.

As shown in the following diagram, the Persistence Manager has several responsibilities. It:

- saves new objects in the database, creating every row in every table necessary to store the state of the object.
- saves modified objects in the database, updating rows that record the state of the changed object.
- materializes objects from the database (that is, creates a new instance of the object and fetches its state information from the rows of the tables that store that state information, and puts the state information into the attributes of the object).
- fetches lists of data for selection by the user.



Optionally, it may also:

- cache objects for faster retrieval.
- notify (via events) other objects that objects it manages have changed.

Fetching Objects

One of the responsibilities of the Persistence Manager is to retrieve objects from the database when they are needed by other parts of the application. This can be single objects, sets of objects, or information about what objects might be available for retrieval.

Fetching Simple Objects

When an object is needed by an application, and that object is a simple object, the Persistence Manager can fetch it directly and return it. In this case, the application provides the value of some attribute that will uniquely identify the object to be retrieved. This could be either a primary key value or an alternate key value. In either case, the database will return a single row with the information.

As an example, suppose the application requests the employee object that has the employee id “47912”. The Persistence Manager issues SQL to fetch the attributes from the Employee table that correspond to 47912. The Persistence Manager then creates an instance of the Employee object and populates its attributes with the values retrieved from the database. (Note: In some cases, it is possible to fetch directly into the object, if it is created in advance.) Then, this object is returned to the caller.

Fetching Complex Objects

Complex objects are objects that 1) have other objects as components, or 2) are associated with other objects in a way that makes sense to have them all together. Examples of the first group are Invoice objects that have Invoice Line Items as components. Examples of the second group might be Customer objects that have associated with them a collection of recent Orders, a collection of recent Shipments, a collection of recent Invoices, a collection of recent Payments, and a collection of Contacts for that Customer.

In either case, when the "top" or "root" object is requested, consideration must be given to how much "neighborhood" is fetched at the same time. At one end of the spectrum is a desire to minimize the number of trips across the network to fetch the information needed for this bit of processing. At the other end of the spectrum, is the desire to minimize the amount of information actually transferred in any single request, because it may not be known what information the user is actually interested in (an expanded discussion of this follows in the "Drill Down" section). Transferring excessive information can result in 3 problems: 1) fetching too many rows from the database, making queries (and their responses) more complex and time consuming than necessary, 2) large amounts of information must be transferred over the network, and 3) these objects must then be cached on the client (not to mention on the server, before they are transferred). These combine to seriously impact performance is the quantity of information being fetched is large enough.

If the processing that is about to be done is known, then the issues are clearer and the decisions are easier to make. Everything that is needed can be brought over in a single request.

However, if there is no a priori knowledge of how the caller will use the additional information which is accessible from the "top" object, then more care must be applied to avoid always fetching too much information, thereby reducing the performance of the system (both from a database and network perspective). Additionally, sometimes a restructuring of the user interface can help in knowing what is needed. For example, if the ability to look at and modify detail about invoices is on a separate screen, then the detail information can be fetched when that screen is entered. (Note: user requirements sometimes make this impossible.)

Fetching Collections of Objects

There are many situations in which the caller will request more than a single object. In each of these situations, there must be a careful balance between fetching things using set queries (which is the best way to get things from a relational database) and fetching more than is needed. There are a number of cases that occur frequently, and each of these is discussed below.

Fetching Lists of Objects

Sometimes, the requirement is to fetch a list of things that is then displayed to the user for further selection. An example of this would be a list of Employees that are presented to the user in a list box, for further selection. Additional processing will then be done to each of the selected Employee objects.

One way to do this would be to fetch, materialize and return to the caller all of the employees that meet the given criteria (which could even be "all of them"). This will result in a significant overhead to create and pass back numerous objects that will never be needed.

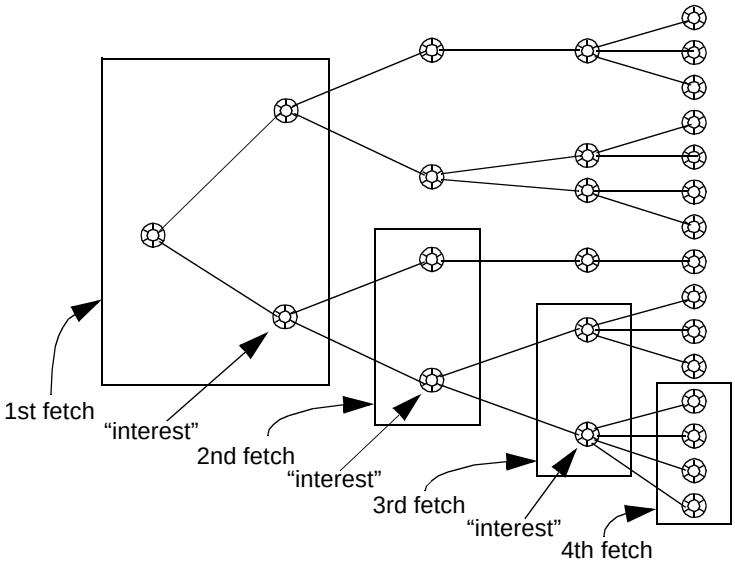
A better way to approach this problem is to fetch all of the employee names (or whatever else would be displayed to the user) and return an array of those. This can be done using a set query (with a fetch loop). This list is then displayed to the user and the desired employees are selected. This (much smaller) list is returned to the Persistence Manager and the appropriate Employee objects are materialized and returned to the caller. Unless almost all of the objects in the original set must be processed, this approach will provide better performance.

"Drill Down"

"Drill Down" occurs when the user fetches an object that has components, and each of its components also has components, and they have components, etc. and wants to select one of the objects to see its components. An example of this might be regions which contain customers, who have placed orders, which contain line items which reference products. Another example might be a top level assembly in a bill of materials structure. In either case, materializing all connected components would result in an extremely large set of objects being returned (certainly hundreds or thousands, potentially tens or hundreds of thousands).

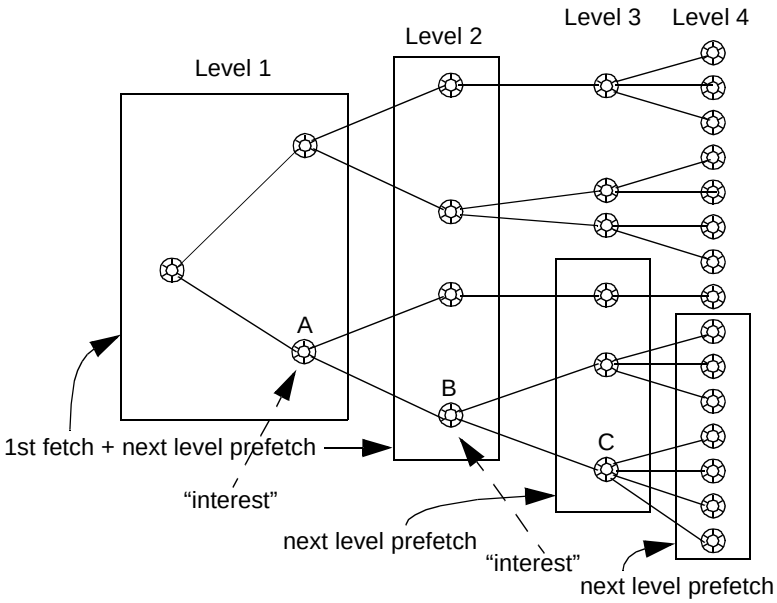
In this case, the entire object graph cannot be fetched in a single call. Not only would performance suffer significantly, it would be for no reason. The very nature of the "drill down" application is to focus, at progressively deeper levels, on a narrower and narrower part of the object structure. Therefore, the vast majority of what has been fetched is irrelevant to the user's interest. Unfortunately, there is no way to predict where the user's interest lies.

Therefore, the fetching strategy that should be used is to bring in the "top" level and present that to the user (the Persistence Manager simply returns the top level; presentation is accomplished by another object). Then, when the user indicates the area of interest, additional objects are fetched from the persistence manager, one level (or possibly a few levels) at a time. This is illustrated below.



In each case, fetches were done only to the next level, then the user expressed interest to allow more selective fetch of the next level down. At any point in time, if the remainder of the graph can be fetched cheaply, or because it is known that everything below that point is needed, then it should all be retrieved in one call.

A variant of this technique occurs when prefetching is done. In this case, some unneeded objects will be retrieved (because they are "prefetched") but it will give the appearance of better performance, because when the user clicks on the next level, it will already be there. This is shown below.



When the Level 1 information is retrieved, the Level 2 information corresponding to all Level 1 objects is "prefetched". Now, when the user expresses interest in "A" (again, through the GUI), all Level 3 descendants of "A" are prefetched while the user examines the Level 2 objects presented as a result of selecting "A". The idea is to get these back to the "front-end" before the user actually identifies the drill down path at "B". Once "B" is identified, all Level 4 descendants of "B" are prefetched, so that when the selection of "C" is made, all lower level objects are present. Again, at any point, if the remaining number of objects is small, they all should be fetched.

How to Fetch Additional Objects from the Complex Object Graph

In all of the cases where there are complex object graphs and only the top portions are being fetched, there will be a point where lower level objects must be fetched. When this time comes, there are several options regarding how to do this. These are:

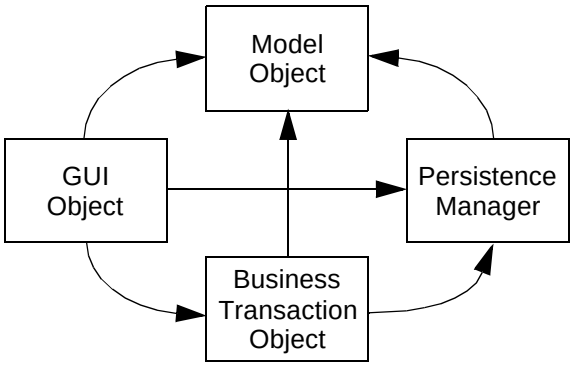
1. An accessor is defined on the model object which contains the SQL to fetch the child objects (at the next level down) in the object graph. (Bad choice - do not do this!)
2. An accessor is defined on the model object which calls the Persistence Manager to fetch the next level of objects in the object graph.
3. The object that "has" the model object (in this case, either a GUI object or a Business Transaction object) calls the Persistence Manager to fetch the additional objects.

The first option will eliminate all of the benefits of the three tier architecture by moving the SQL out of the Persistence Manager and into the Model Object. Under no circumstances should this be done. The second and third options are more reasonable solutions to this problem, so a more thorough look at each is warranted.

Option 2 provides slightly better encapsulation, requiring the GUI or Business Transaction object to call an accessor which then, upon determining that the component objects

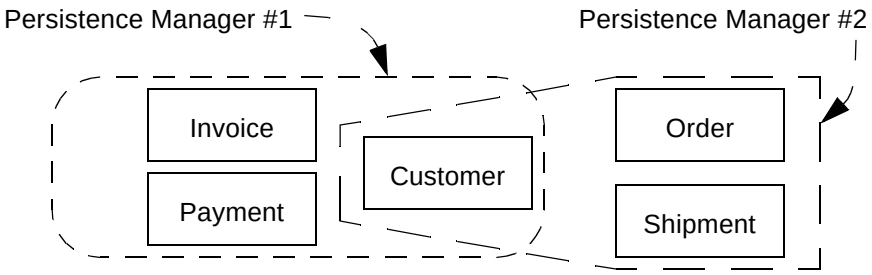
are not present, calls the persistence manager to fetch them. This minimizes the need for either the GUI or the BT object to know whether the components are present or not. However, this creates a software engineering problem.

Consider the dependency graph for the objects before this type of call is introduced. It looks like this:

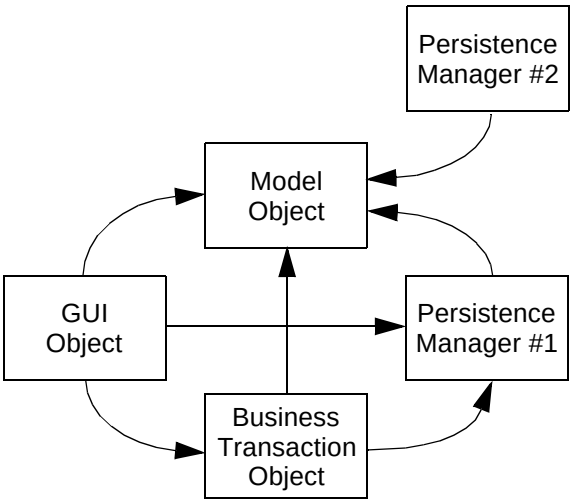


Because of this graph, each of the different types of objects shown above can be put in a different project, if desired.

Also, a model object may be used by more than one Persistence Manager. This would happen if a model object were shared between various parts of an overall application. An example of this would be a Customer object which might be used by a Persistence Manager dealing with Orders and Shipments, as well as a different Persistence Manager dealing with Invoices and Payments. This is shown below.

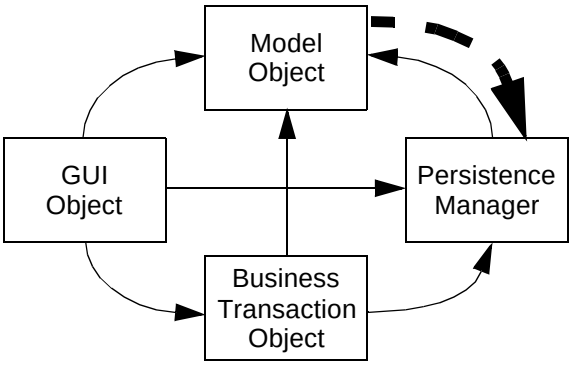


In this case, there would be a dependency graph like this:



It is still possible for each of the objects to be in a separate project.

Now, consider the case where the model object calls the Persistence Manager to get the additional component objects. This produces a dependency graph like this:

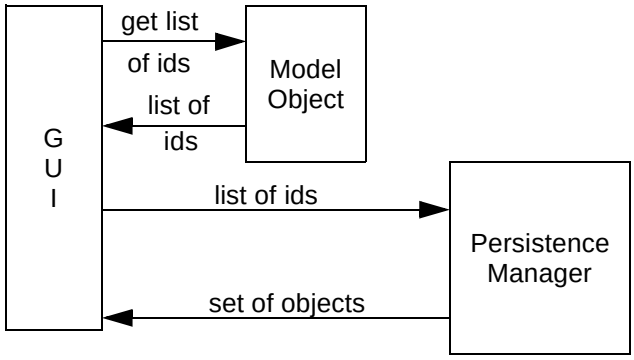


The mutual dependency between the model object and the Persistence Manager creates a peer-to-peer relationship. This means that the model object and the Persistence Manager must be in the same project. Extending this to the situation above (multiple Persistence Managers, one for each application area) requires the new persistence manager to be in the same project as the model object as well. The extended result of this is that all model objects and all Persistence Managers must be in the same project (unless there are completely disconnected portions of the dependency structure). This leads to significant problems as the size of the project grows.

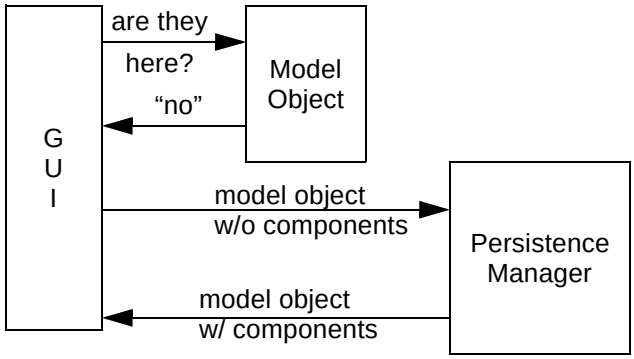
Consequently, for software engineering reasons, this solution is rejected.

Option 3 allows significantly better software engineering, at the expense of some knowledge of the object graph on the part of the GUI object or Business Transaction object. In this case, that object (for the purposes of this example, the GUI object will be used - the same issues apply with respect to the Business Transaction object) must do something to get the component objects from the Persistence Manager.

One option is to have a list of identification attributes present in the model object. The GUI object then requests this list and sends it to the Persistence Manager to have those objects fetched. An object interaction graph of this would look like this:



This requires that identification attributes be present in the component object. While this is sometimes possible, it is by no means universal. A better way to retrieve the component objects is to determine if thy are present and, if not, pass the model object to the Persistence Manager and request that the component objects be fetched. The model object is returned with all of the component objects in place. The object interaction graph for this approach looks like this:



The (pseudo) code to process this would look like this:

```
if modelobj.componentsNotPresent() then
      PersistenceMgr.fetchComponents (modelobj);
      end if;
```

An alternate (but equally effective implementation) would be:

```
if (compList = modelobj.getComponents() == NIL) then
      PersistenceMgr.fetchComponents (modelobj);
      compList = modelobj.getComponents();
      end if;
```

When You Get Too Much

When fetching sets of rows from the database, the Persistence Manager must always be prepared to deal with getting more than expected. This is particularly true when the user provides data values that qualify the query. For example, the query:

```
SELECT firstname, phone FROM person
     WHERE lastname = 'glockenheimersberg' AND
           city = 'East Podunk';
```

Is likely to return few rows (if any). If the user is allowed to enter the last name and the city to be used for the search, consider what happens when they enter:

```
SELECT firstname, phone FROM person
     WHERE lastname = 'smith' AND
           city = 'London';
```

and a few million rows come back.

At this point, the Persistence Manager has several choices. The easiest (and perhaps least helpful for the user) is to abort the query and force the user to requalify. Perhaps they meant "London, CT", and would have gotten less rows.

Another option is to return some number of rows, and fork a thread to continue retrieving rows in groups. A logical grouping is whatever fits on a screen. This case, however, requires additional coordination between the Persistence Manager and the "consumer" of the objects returned.

Saving Objects

Under this general title comes both the initial saving of newly created objects and the updating of objects that have changed during the course of a transaction.

Saving Simple Objects

Saving simple objects can be accomplished with a single call to the Persistence Manager to save the object in the database. The object is passed to the Persistence manager and the necessary tables are updated (or rows created) to save the data from the object. In general, a simple object should be only updating one or two tables in the database. Thus, performance should not be a significant issue for these.

Saving Complex Objects

When a very complex network of objects (generally, this is an object graph with many component objects) is returned to the Persistence Manager so that it can be updated in the database, there are a number of options regarding how the object is updated.

Initially, the options are to save either the entire object graph or simply the part(s) that have been updated. (Obviously, if the object is being created, the entire object must be saved in the database.) While the entire object could be saved, doing so often requires significantly more database access than just updating the parts that have changed. This can reduce performance, in some cases significantly.

A better approach is to update only the portions that have changed. However, this requires some coordination between the Persistence Manager and some object who knows what has changed. This other object could either be the object that is passing the model object to the Persistence Manager (either the User Interface object or the Business Policy object) or it can be the model object itself.

UI or Business Policy Object Controls Update

If the calling object is making the determination, there are two ways that this can be accomplished.

The calling object can call a single method that is passed the complex object to be updated and an additional argument that indicates what has been changed and, consequently, what must be updated. Unfortunately, this violates a well-known software engineering principle that an argument to a calling routine should not control the path of execution in the called routine. (Violating this produces an undesirable coupling of code that is a source of frequent and irritating errors during system enhancement.)

The second way this can be done is to have the calling object invoke a different method (defined in the Persistence Manager) for each type of update desired. Actually, there are two different ways to accomplish this as well.

The first way is to have the caller call a method on the Persistence Manager for each part of the complex object that changed. Thus, if four different parts of the object changed, four different calls would be made to different methods of the PM. In most systems, this would result in four remote calls to the Persistence Manager, a low performance way to implement saving.

Another variation of this is to combine the individual methods into the various combinations that will be called based upon the groups of subparts that may be modified together. This alleviates the necessity of calling multiple methods on the Persistence Manager, but requires the identification of every combination of methods that may be required and the definition of a corresponding method. This results in an unnecessary creation of methods on the Persistence Manager.

Either of these techniques can be obnoxious for the developer who must keep track of what part changed and call the corresponding method(s) of the PM. Furthermore, it is not good software engineering to have the caller determine (potentially in hundreds of places) what changes have been made and which methods should be called. If later

modifications are made to the application structure, the effort of consequent code modifications could be quite large.

Model Object Controls Update

A better approach would be to simply pass the updated object to the Persistence Manager and request that it be saved. Now, the question of how the Persistence Manager knows what has been changed arises. The best way to accomplish this is to ask the complex object that was changed. There are two fundamental ways that this can be accomplished.

In the first approach, some sort of code (perhaps an integer) is stored in the object to reflect what needs to be saved. While this does put the responsibility for knowing what changed in the right place (in the object itself), it uses what may end up being an obscure code to communicate with the Persistence Manager.

A better approach is to add a number of boolean methods to the complex model object that each report whether a certain portion of the object has been changed (and needs to be updated). Then, the Persistence Manager can have a series of interrogations of the model object like the ones shown below:

```
if modelobject.part_a_changed() then
      fetch_and_save_part_a()
if modelobject.part_b_changed() then
      fetch_and_save_part_b()
if modelobject.part_c_changed() then
      fetch_and_save_part_c()
```

So, this is the approach that should be used for all complex model object updates.

Implementation Details

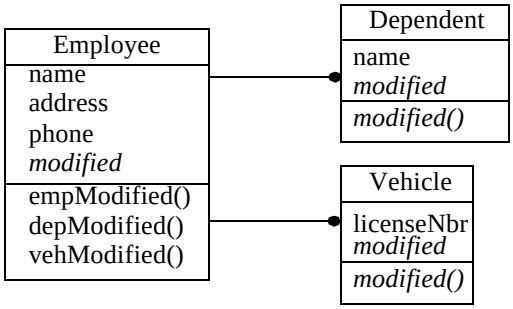
The actual implementation of this approach is quite simple. It uses a combination of inheritance and delegation to provide a “framework” type solution for ease of implementation system-wide.

First, at the highest level in the application object hierarchy, a boolean value is defined that indicates whether an object has been changed. Further, a public method is defined at this level to return the value of this attribute. The method (obviously) returns a boolean value.

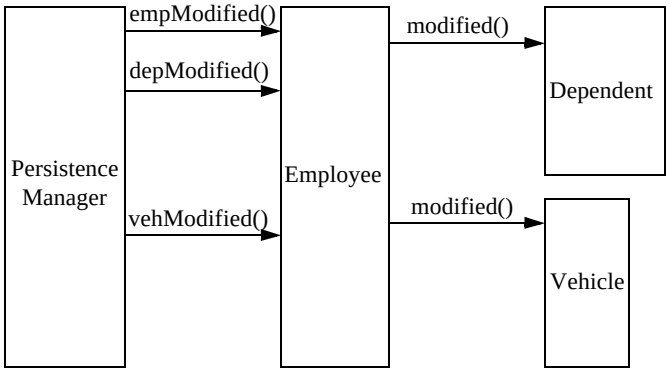
Whenever an object is created (first time in existence anywhere), the “modified” attribute is set to “true” to indicate that it needs to be saved. Whenever the object is materialized (read in from persistent storage), the value of this attribute should be “false”. Whenever the object is changed, it sets the value of this “modified” attribute to “true”.

Now, whenever a request is made to save a complex object, it provides the methods described above to report whether various portions need saving. These methods query the component objects to see if they have been modified, and report the answer. This can be done for each of the component objects. An example is shown below.

Consider the following object model:

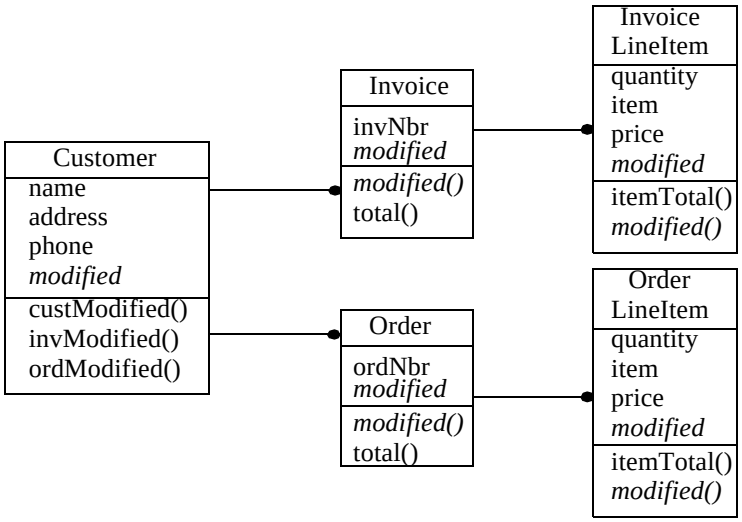


In this example, Employee is the complex object with the components Dependent and Vehicle. (Note: the attributes and operations that are shown in italics are not actually defined on the classes as shown; they are inherited from the superclass.) Each of the components in turn queries its components to see if they have been modified. This yields the object interaction diagram shown below.

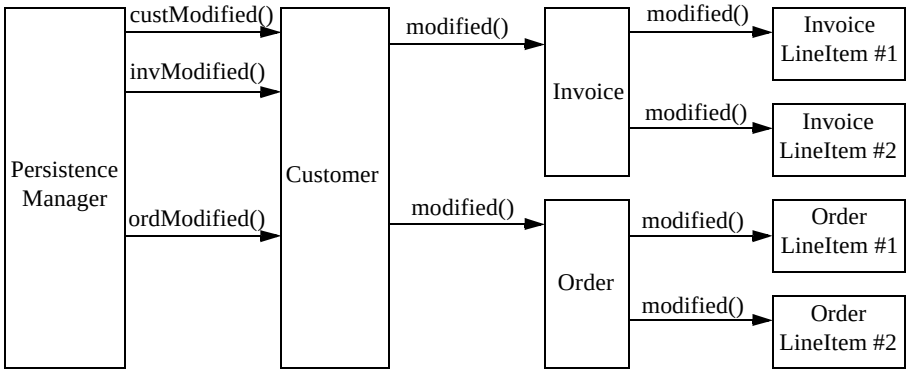


In this example, the Persistence Manager sends a message empModified) to the Employee object to see if the basic employee information has changed. If so, the basic Employee information (in this case, name, address and phone) is fetched and saved. Then, another message is sent (*depModified*) to see if the Dependent object has been modified. In order to determine this, the Dependent object will check its own "modified" attribute. If it has changed, then *depModified* returns "true" and the Persistence Manager saves the Dependent. If it has not been modified, *depModified* returns "false" and the Persistence Manager does not need to save the dependent. Then, the last message (*vehModified*) is sent to the Employee object. The Employee object again sends the modified message to the Vehicle to see if the Vehicle object has been modified. If so, it returns "true" to the Persistence Manager, and the Persistence Manager saves the information associated with the Vehicle object.

Now, let's consider a more complex example.



In this example, Customer is the complex object with the components Invoice and Order. (Note: the attributes and operations that are shown in italics are not actually defined on the classes as shown; they are inherited from the superclass.) In this more elaborate example, each of the components in turn queries its components to see if they have been modified. This yields the object interaction diagram shown below.



In this example, the Persistence Manager sends a message (*custModified*) to the Customer object to see if the basic customer information has changed. if so, the basic customer information (in this case, name, address and phone) are fetched and saved. Then, another message is sent (*invModified*) to see if the invoice object has been modified. In order to determine this, the invoice object will check its own "modified" attribute, and then ask each component line item if it has been changed. If any of those have changed, then *invModified* returns "true" and the Persistence Manager saves the invoice. If none of them has been modified, *invModified* returns "false" and the Persistence Manager does not need to save the invoice or its components. Then, the last message (*ordModified*) is sent to the Customer object. The processing is essentially the same for Invoice.

Another benefit to this style of saving objects is that the granularity can be adjusted as required. In this example, the level of granularity was at the components of Customer (Invoice and Order). With small changes, it can be adjusted to go down to the next lower level, the subcomponents. In this case, for Invoice and Order, those subcomponents would be the individual line items. Depending upon the application, this could be essential.

Internal Collaborations

While it is possible to write a single monolithic Persistence Manager, from a software engineering perspective, this is an undesireable thing to do. It defeats one of the biggest advantages of the object paradigm - minimizing the impact of change.

Conclusion

Each situation of saving and retrieving objects in the Persistence Manager needs to be examined and the performance issues dealt with when the interaction is planned. Transferring unnecessary amounts of information can result in decreased performance, so care must be taken to design each interaction to transfer no more than necessary, but to transfer all of that in a single access.