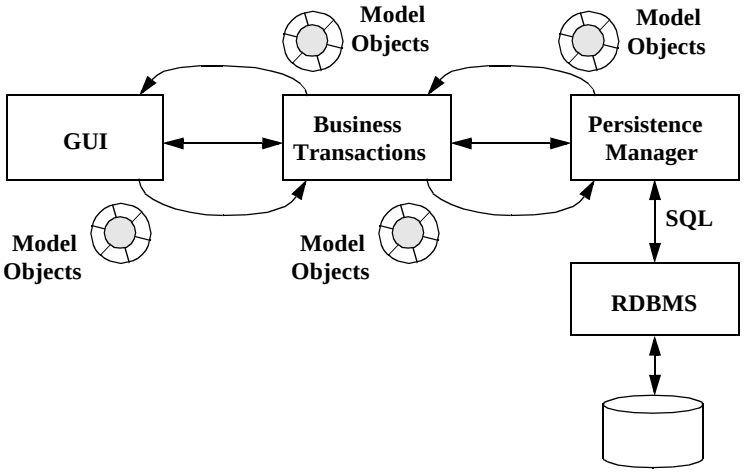
3-Tier Architecture - An Introduction

Stephen McHenry

An introduction to building systems using a 3-tier architecture.

This document describes how to implement software systems using a 3 tier architecture to maximize deployment flexibility.

How It Works



Each of these layers has a certain set of responsibilities. The responsibility of the GUI is to interact with the user and initiate the transfer.

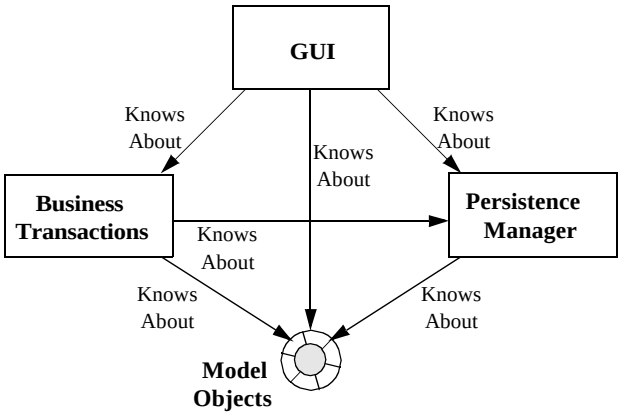
The responsibility of the Business Model Objects is to encapsulate the appropriate abstraction from the problem domain. Typically, these are the objects that "jump out" at designers when examining a problem domain. They are almost always persistent objects in the system.

Their persistence, that is - getting them into and out of the database, is the responsibility of the Persistence Manager. It is the only portion of the system that should contain any SQL.

The responsibility of the "Business Transaction" is to encapsulate all of the processing logic and business rules that don't go anywhere else. Typically, these would be an encapsulation of the business policy as it relates to this particular transaction. This type of object is relatively new in the overall growth of object technology, and has been given a myriad of names depending upon the "inventor". Among those names are: control objects, business transaction objects, business policy objects and policy manager objects. By any name, their intended function is the same.

Dependency Graph

Incidentally, note the dependency graph that this creates.

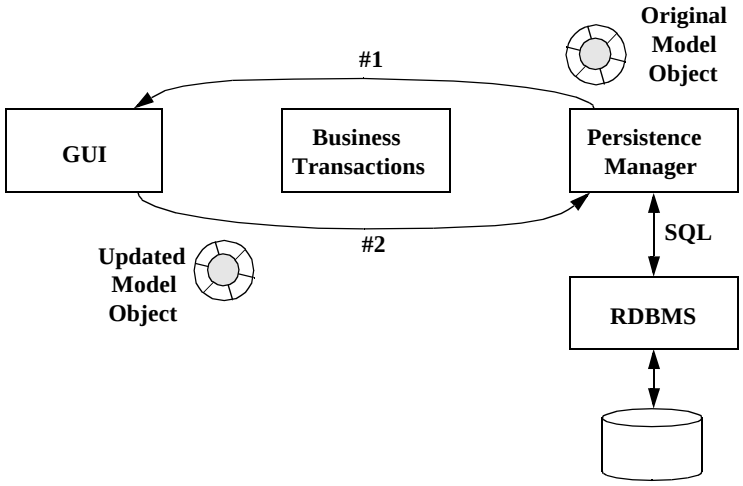


Note that it is acyclic (that is, it does not contain cycles, which would represent circular dependencies).

Changing an Employee

Now, let's examine how this architectural approach might be applied in the context of some function in the system. In this example, the GUI might need to modify some object that already exists in the database. An example of this might be changing the hair color of an employee.

In this simplest of cases, the original diagram (which is a "prototypical" interaction) would actually look like this:



The interaction of each tier would then be:

GUI	Business Transaction	Persistence Manager
• User enters the information to fetch the employee to be changed • Calls Persistence Manager to fetch employee		
		• Issues SQL to database to fetch each “employee” row • Materializes employee objects • Passes employee object back to GUI
• Modifies the employee object • Sends the employee object back to the persistence manager and requests that it be saved		
		• Fetches the attributes that must be updated from the employee object • Issues SQL to the RDBMS to update the employee object

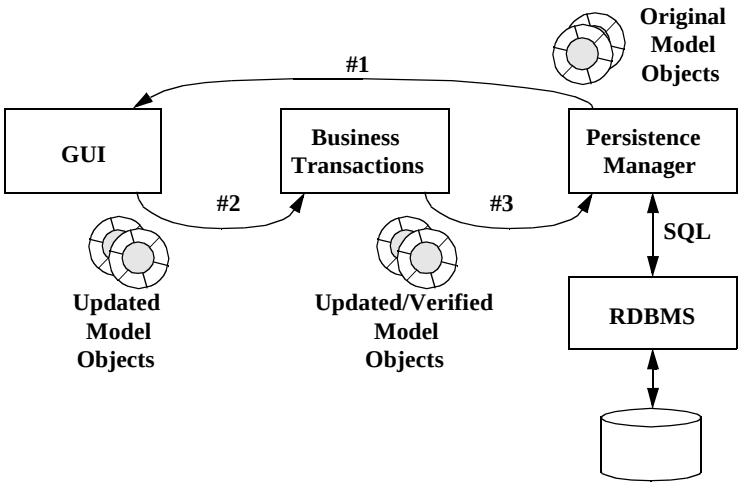
GUI	Business Transaction	Persistence Manager
		• Returns “success” to GUI
• Commits the work • Notifies the user of completion		

In this example, there were no business rules that needed to be applied to changing of a person’s hair color, so the GUI was able to interact with the persistence manager directly, initially to fetch the employee and later to save it.

Modifying an Invoice

In a more complex example, it might be possible for the GUI to fetch the object directly from the persistence manager, but changes to it might need to be validated to be sure they do not violate business policies. If this is the case, the object will be returned to a Business Transaction object (typically a shared service in the environment), for checking and then saving via the persistence manager.

An example of this might be to modify an invoice that already exists. In so doing, business rules need to be checked when items are added to be sure the customer’s credit limits are not exceeded, and when items are removed, to be sure that appropriate minimums for the discount levels have not been violated. An example of this is shown below.



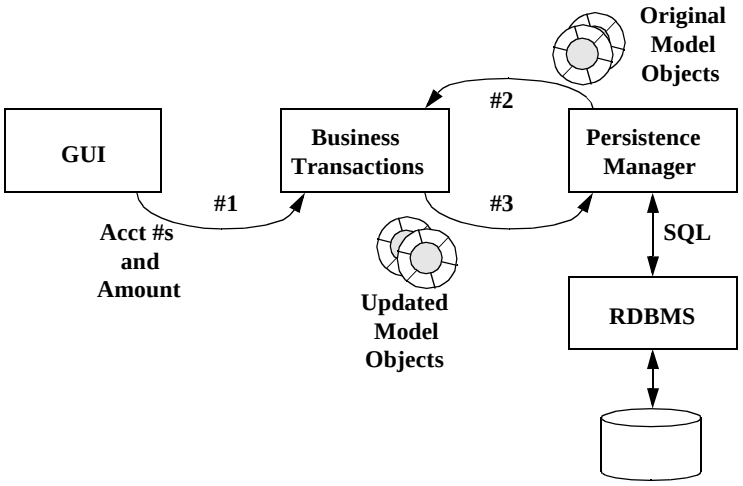
The interaction between layers is now:

GUI	Business Transaction	Persistence Manager
• User enters the order number to be modified • Calls Persistence Manager to fetch the order		

GUI	Business Transaction	Persistence Manager
		• Issues SQL to database to fetch the "order" row and its associated line items • Materializes all objects • Passes all objects back to GUI
• Modifies the order • Sends the order object back to the Business Transaction service and requests that it be saved		
	• Applies the necessary business rules to validate that the order has not been improperly modified.	•
	• Sends the order to the Persistence Manager to save it	•
		• Fetches the attributes that must be updated from the order object • Issues SQL to the RDBMS to update the order object • Returns "success" to Business Transaction
	• Commits the work • Returns to the GUI	
• Notifies the user of completion		

Move Money Between Accounts

Now, let's look at a simple transaction which transfers money from one account to another. Here is how the collaboration might work between each of these layers:



GUI

- User enters the account numbers and the amount
- Calls Business Transaction object (service) and passes the account numbers and the amount

Business Transaction

- Calls the Persistence Manager to fetch the two accounts from persistent storage
- Modifies the “from” account
- Modifies the “to” account

Persistence Manager

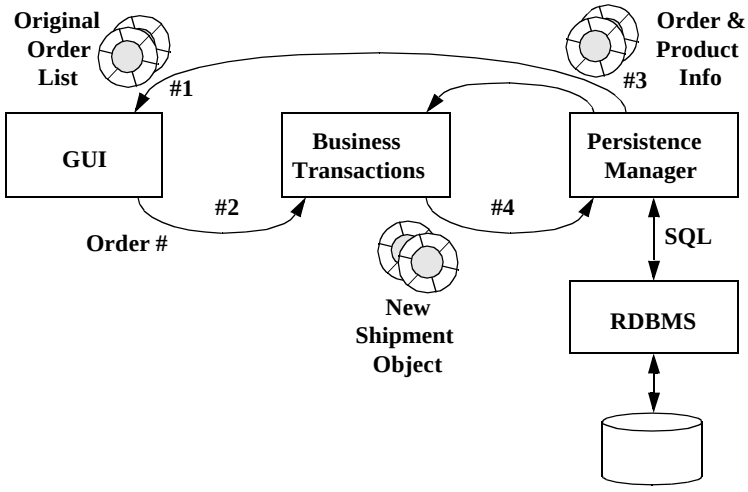
- Issues SQL to database to fetch the “account” rows
- Materializes both account objects
- Passes both accounts back to Business Transaction Object

GUI	Business Transaction	Persistence Manager
	• Sends the account objects back to the Persistence Manager and requests that they be saved	
		• Fetches the attributes that must be updated from the account objects • Issues SQL to the RDBMS to update the account objects • Returns "success" to Business Transaction
	• Commits the work • Returns to the GUI	
• Notifies the user of completion		

Notice that the GUI simply collected the information necessary to allow the Business Transaction object to do its job, namely, performing the transfer.

Turn Order Into Shipment

In the final example (and most complex one presented here), the user wants to turn an order into a shipment. In this case, the GUI talks directly to the persistence manager to fetch a list of orders from the database, the user selects the order to be shipped, and that order id is passed to a business transaction object. Now, this business transaction object must fetch the entire order and information about all of the products to be shipped, and then create a shipment which is then saved.



GUI	Business Transaction	Persistence Manager
• User requests a list of all orders that can be turned into shipments from Persistence Manager		
•		• Issues SQL to database to fetch the list of orders
•		• Materializes an array object containing the list of orders
•		• Passes array of orders back to GUI
• User selects the order • Calls Business Transaction object (service) and passes the account numbers and the amount		
	• Calls the Persistence Manager to fetch the order and product objects from persistent storage	
		• Issues SQL to database to fetch the “order” row • Materializes order objects • Passes order object back to Business Transaction Object
	• Creates a shipment object • Calls Persistence Manager to fetch product information for each line item	
		• Issues SQL to database to fetch the “product” row(s) • Materializes product(s) objects • Passes all objects back to Business Transaction Object
	• Add product(s) to the shipment	

GUI	Business Transaction	Persistence Manager
	• Sends the shipment object back to the Persistence Manager and requests that it be saved	
		• Fetches the attributes that must be updated from the shipment object • Issues SQL to the RDBMS to update the shipment object • Returns "success" to Business Transaction
	• Commits the work • Returns to the GUI	
• Notifies the user of completion		

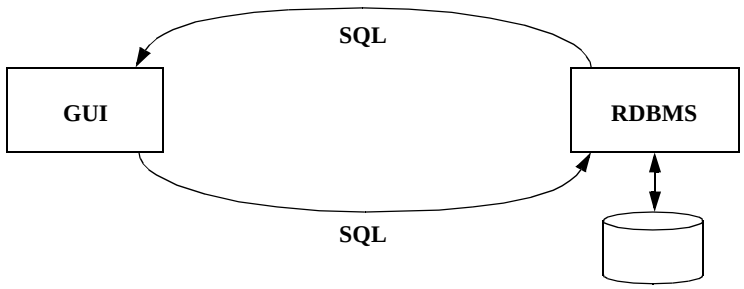
In this case, the GUI fetched one set of information (the list of orders) while the Business transaction object (service) fetched a single order (but this time, all of the order information).

Alternate (less good) Approaches

Having looked at several ways of utilizing the 3-tier architecture, now let's look a a few ways of structuring the system that are not as good. These range from the truly atrocious to just somewhat bad.

The Worst Way

The very worst way to structure the architecture of a system is to embed SQL directly in the "go" button of the user interface. This results in the structure shown below:

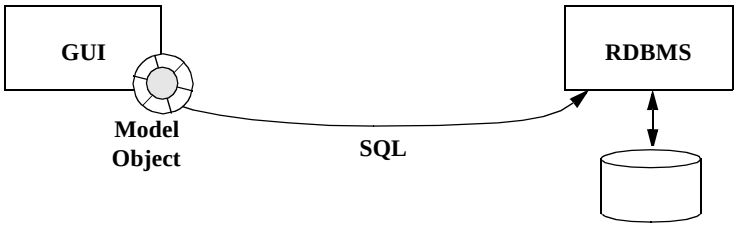


With this type of architecture, the system is forever limited to a 2-tier structure. A distributed object system is not needed to develop this; Visual Basic or Powerbuilder is quite adequate. However, this will not scale to a large number of users.

Still Bad

A common solution that seems to be intrinsically attractive to many designers is to put the SQL to save or fetch a Business Model object in the object itself. This is often put in a method called “save”. When “save” is called, the object issues SQL to save itself in the database.

Unfortunately, this can create the same problem as the preceding example - issuing SQL from the client partition. An example of this is shown below.



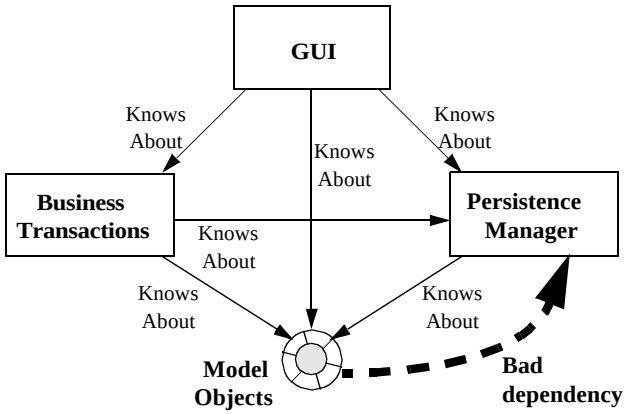
In the above example, the object is shown attempting to save itself.

Now, consider the problem when we need access to an object. Presumably (to be consistent with the location of the SQL for saving) the location of the SQL to retrieve an object should be in a method called “retrieve”, also defined on the same object. However, this now presents an interesting dilemma - in order to fetch itself, it must exist (in one sense), but it doesn’t exist (in another sense) because it hasn’t been fetched yet. Furthermore, if you want customer # 123, where do you put the “123”? In the customer object before it is fetched? On the surface, this solution appears desirable, but upon detailed examination, its flaws abound.

A Little Bit Better (at least philosophically)

The next logical step to those that intrinsically like the object saving itself is to require the object to call the persistence manager when told to save itself. Thus, the object will not need to contain the SQL (this can be placed inside of the Persistence Manager, as in the original solutions) but the object itself still controls its “saving” destiny.

The problem created by this approach is that it now creates a new, undesirable coupling between the object itself and the Persistence Manager. This is shown in the dependency graph below.



So, now the Persistence Manager must know about the model object (in order to save it and fetch it) and the model object must know about the Persistence Manager in order to call it to be saved and (perhaps) fetched.

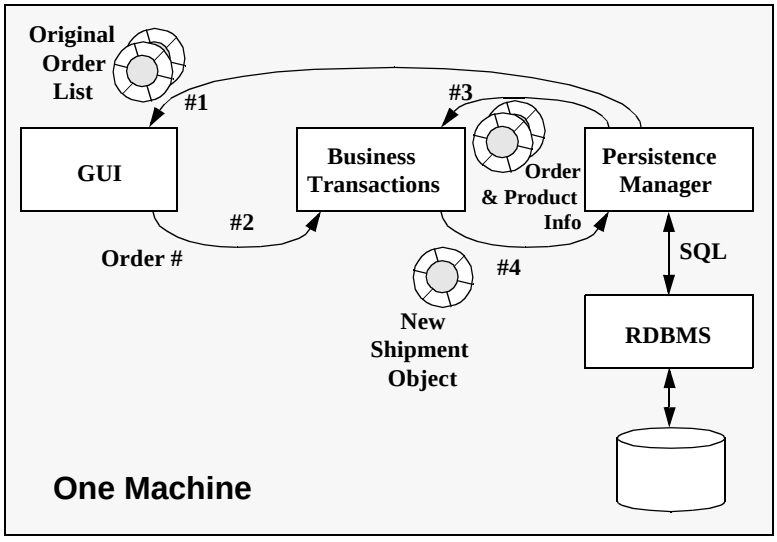
This has two negative side effects. First, it creates a peer-to-peer relationship between any model object and its Persistence Manager. Second, (and as a result of this) the model object and the Persistence Manager must now be kept in the same Project, in the Forte environment. In an ideal world, there might be a Persistence Manager for every different type of model object, but in reality this never happens. So, if there is only one Persistence Manager, and there are 250 types of model objects, these must all be in the same project. This is very undesirable was well. Even if some natural clustering can be devised, there still may not be more than 4 or 5 Persistence Managers which would mean an average of 50 to 60 model classes with a Persistence Manager.

However, this implies a more fundamental flaw in the overall strategy - namely, that everything that must be done to an object, the object should know how to do to itself. Thus, the object should know how to display itself, how to print itself, how to archive itself, and so on. We already know that objects are not necessarily the best ones to determine how they are displayed (hence, the notion of multiple “views” of a single object). The same principles apply to saving and restoring as well.

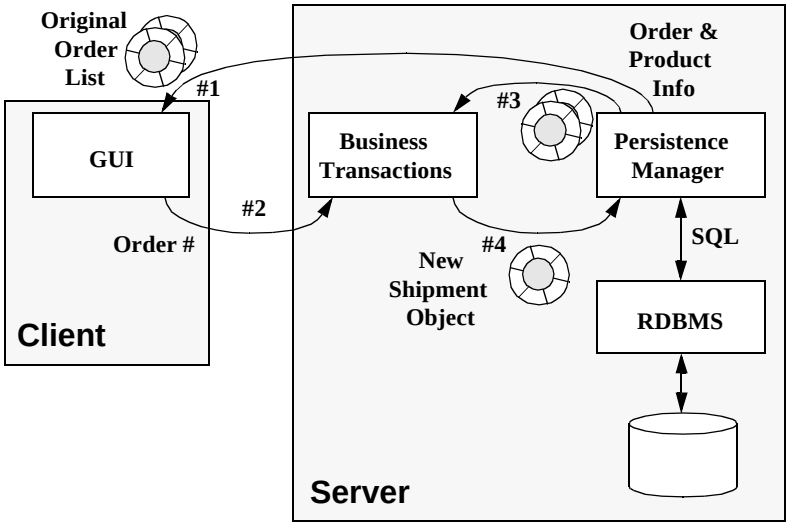
Deployment Considerations

When a 3-tier architecture is used, there are a number of options that become available at deployment time. Another advantage is that the choice of which option is selected may be deferred until the system is actually deployed. Furthermore, the system may be reconfigured to provide better performance, even after it has been initially deployed.

One of the deployment options is to have all of the tiers on a single node. This choice is almost never selected for actual deployment of an operational system, but it is frequently used during development. This is shown below.



Another option is to split the functionality between a "client" system and a "server" system. When this option is chosen, there is a choice as to where the "middle" tier (containing the business logic) is located. One choice is to put it on the server node. This is shown in the following diagram.



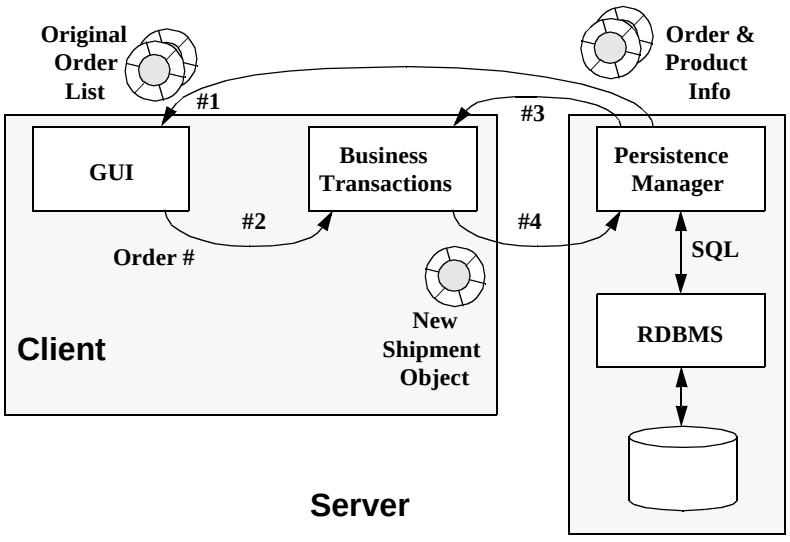
Using this approach (sometimes known as "fat server"), all of the business processing is done on the server. So, it is quite common to find the server quickly overloaded if there are a large number of clients (hundreds or thousands). The ultimate "fat server" is a mainframe.

Another choice is to put it on the client node. This choice is shown in the following diagram. This approach is sometimes known as “fat client”. One of the often touted advantages of this approach is that it allows the power of the desktop machine to be used to off-load processing form the server. Indeed, many client machines are now driven by very powerful processors and this approach can help to more effectively utilize that previously unused processing power.

However, there is a significant disadvantage to this approach. Every client must be sufficiently capable to be able to handle all of the GUI processing (a job that truly belongs on the client) plus all of the logic for the business processing. In many cases, this requires these machines to be upgraded from (say) 8 or 16 MB of memory to (say) 32 or even 48 or 64 MB of memory. Similar expansion of disk space may also be required. If the deployment environment has hundreds or thousands of client machines that need to be upgraded, this can be a significant expense.

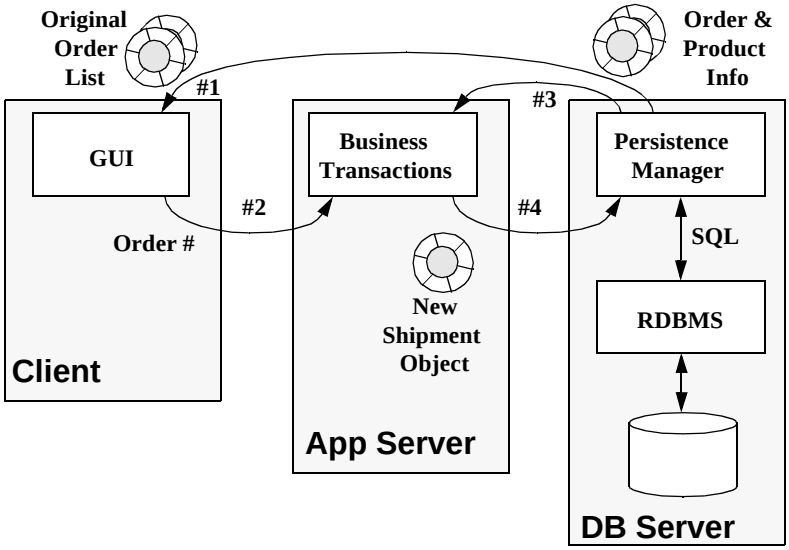
There are additional costs that must be considered as well, such as installing and testing this additional hardware in each of the client systems. This can result in a formidable workload for an IT department.

Also, because of the cost, there is the tendency to buy the minimum amount of memory and disk that can handle the load. As later versions are released, the requirements may expand, necessitating additional memory and disk. This means a second (and perhaps a third) round of installation is necessary.

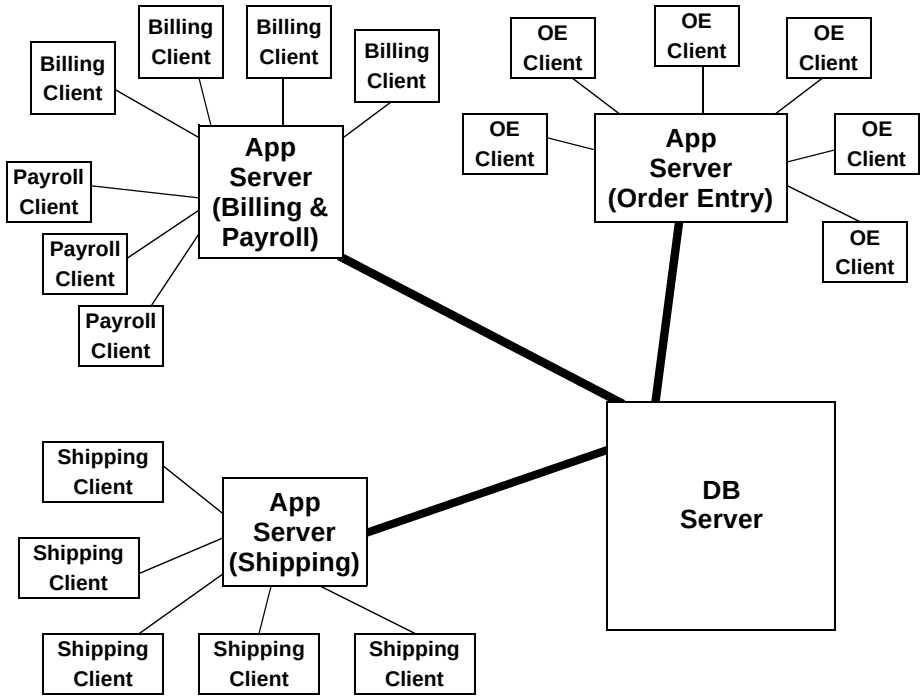


When using traditional client-server architectures, the options have now been exhausted. Using a 3-tier architecture, there is another approach that works very well. This is to add a mid-level application server between the client and the database server. This application server contains the business transaction processing to support a smaller number of users. Typically, these servers are departmentally oriented (payroll, billing, shipping, etc.).

Using this approach, it is possible to better share resources (memory, disk and processing power) between a group of users. This approach is shown in the following diagram.



This approach allows construction of physical network topologies similar to the one shown in the following diagram. Note that an additional benefit is that, if network traffic becomes a problem, each of the application servers can be placed on a separate network to further increase performance.



Conclusion

In this paper, we have examined a correct way (along with four different examples) of the interactions between model objects, GUIs, Business Transaction objects (services) and the Persistence Managers. When developing real projects, most interactions should fall into one of these main categories. Also examined were a few less desirable ways of structuring the system that result in less optimal architectures.