What is the 3-Tier Architecture?

Posted on 14th October 2012 by Tony Marston

Amended on 11th March 2021

Introduction

This article is in response to N-Tier Architecture - An Introduction written by Anthony Ferrara which I feel gives only a very limited view of what can be a complex subject. Different people have different ideas on how an application can be split into tiers, and different ideas on the benefits which can be gained from making such a split. I have been designing and building multi-tiered applications in different languages since the late 1990s, so my exposure to this subject has been quite extensive. I would like to share with you my experiences on this subject and hopefully give you a more detailed and more accurate picture.

Definitions

In the book Applying UML and Patterns, which was first published in 1997, the author Craig Larman writes the following:

> One common architecture for information systems that includes a user interface and persistent storage of data is known as the **three-tier architecture**. A classic description of the vertical tiers is:
> 1. **Presentation** - windows, reports, and so on.
> 2. **Application Logic** - tasks and rules which govern the process.
> 3. **Storage** - persistent storage mechanism.
>
> The singular quality of a three-tier architecture is the separation of the application logic into a distinct logical middle tier of software. The presentation tier is relatively free of application processing; windows forward task requests to the middle tier. The middle tier communicates with the back-end storage layer.

> This architecture is in contrast to a **two-tier** design, in which, for example, application logic is placed within window definitions, which read and write directly to a database; there is no middle tier that separates out the application logic. A disadvantage of a two-tier design is the inability to represent application logic in separate components, which inhibits software reuse. It is also not possible to distribute application logic to a separate computer.
>
> A recommended multi-tier architecture for object-oriented information systems includes the *separation of responsibilities* implied by the classic three-tier architecture. These responsibilities are assigned to software objects.

Note here that he created a direct link between the *separation of responsibilities* and the 3-Tier Architecture some 6 years before Robert C. Martin published his first article on the Single Responsibility Principe (SRP) in 2003.

Here are some links to other articles on the 3-Tier Architecture:

- Three-Tier Architecture from the Linux Journal
- Three-tier architecture from wikipedia.org
- PresentationDomainDataLayering by Martin Fowler
- 3-Tier Architecture: A Complete Overview from jinfonet.com
- What does 3-Tier architecture mean? from techopedia.com
- How to Organize Application Code With 3-Tier Architecture? By Pavel Kukhnavets
- 3-Tier Architecture from ADL Data Systems
- Client/Server and the N-Tier Model of Distributed Computing from n-tier.com

Note that in Martin Fowler's PresentationDomainDataLayering he starts off with just three layers - Presentation, Domain, Data - but then he complicates it by adding in a service layer and a data mapper layer. I have never seen any need for these last two, which is why I leave them out entirely.

You should note that this architectural pattern was originally designed for compiled languages which ran on a desktop with a bit-mapped display where each movement of the mouse across the screen fired a trigger in the application so that the application could respond to that mouse movement. In web applications this is totally different as the application builds a screen by sending an HTML document to the client's web browser, and there is no communication between the browser and the application until a SUBMIT button is pressed, which causes the contents of that HTML document to be transmitted (or posted) to the application server..

What is a "tier"?

A "tier" can also be referred to as a "layer". A wedding cake is said to have tiers while a chocolate cake is said to have layers, but they mean the same thing.

In the software world Tiers/Layers should have some or all of the following characteristics:

- Each tier/layer should be able to be constructed separately, possibly by different teams of people with different skills.
- Several tiers/layers should be able to be joined together to make a whole "something".
- Each tier/layer should contribute something different to the whole. A chocolate layer cake, for example, has layers of chocolate and cake.
- There must also be some sort of boundary between one tier and another. You cannot take a single piece of cake, chop it up into smaller units and call that a layer cake because each unit is indistinguishable from the other units.
- Each tier/layer should not be able to operate independently without interaction with other tiers/layers.
- It should be possible to swap one tier/layer with an alternative component which has similar characteristics so that the whole may continue functioning.
- Although it is usual to run all of these layers on the same server, there may be benefits from running each layer on its own server.

The idea of being able to swap components in one layer without having to modify components in other layers has enormous benefits. For example, you may start with a presentation layer component which extracts data from a business layer component and formats that data into HTML, but later on you add additional presentation layer components to format the data into CSV or PDF. These additional presentation layer components should be handled without having to make any changes to the business or data access layers. Similarly, if you start with a data access object which handles all the communication with a MySQL database it should be possible to switch to an alternative component for a different database engine, such as PostgreSQL, Oracle or SQL Server, without having to make changes to any components in the business or data access layers.

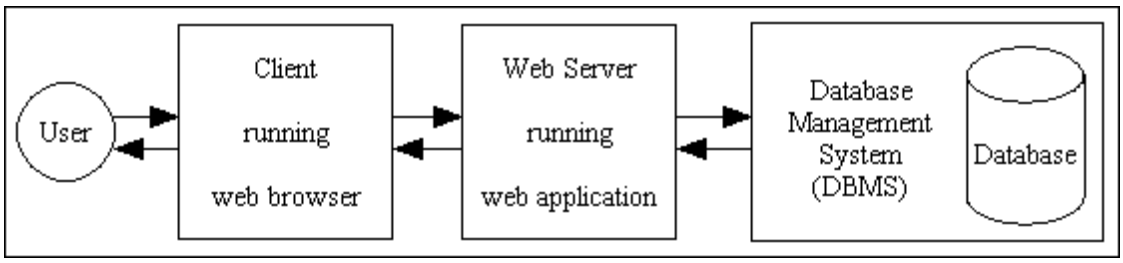
What is the difference between "N Tier" and "3 Tier"?

There is no difference. Some people refer to the N Tier Architecture where 'N' can be any number. I personally have found no use for any more than 3 tiers, which is why I always call it the 3-Tier Architecture. If you try to build an application with more than three layers then be aware that it may have a serious impact on performance, as discussed in Application Performance and Antipatterns.

What the 3 Tier Architecture is not

Some people consider that a web application is automatically 3 Tier as it has 3 separate components, as shown in figure 1:

Figure 1 - A Web Application



Although this would appear to satisfy the conditions outlined in What is a "tier"?, I feel that it fails on one important aspect. Just as a layer cake is made up of several layers of cake - you cannot combine a single piece of cake, a cake stand and a decoration and call it a "layer cake" - you require several layers of "cake". In order for your application to be called multi-layered it is the application on its own - which excludes all external components such as the browser and the database - which must be broken down into several layers. I am discounting everything outside of the application itself for the following reasons:

- The web browser can operate independently of the application, therefore it is not part of the application. Although it is possible for application code to be executed in the client's browser, such as JavaScript, Flash, or ActiveX controls, their usage is entirely optional and can be disabled.
- The database server can also operate independently of the application, therefore it is not part of the application. Although it is possible for application code to be executed in the database, such as database triggers and stored procedures, their usage is entirely optional and can be disabled.

What parts of an application can be split into layers?

In the above example both the web browser and database server are external pieces of software which are totally independent of the application, but the application must contain code to handle the communication with these external objects.

- The application receives requests from and sends responses to the web browser using HTML documents, and it is the code which deals with these requests and responses which can be separated out from the business logic.
- The application sends requests to the database server by generating SQL queries and receiving responses to those queries, and it is the code which deals with these requests and responses which can be separated out from the business logic.

You cannot just take the source code for an application, chop it into parts and call each part a layer. You have to identify specific areas of responsibility, then identify the code which performs within each of those areas. As I said previously, I do not recognise any more than 3 areas, and those 3 are:

1. Presentation logic - the user interface (UI) which displays data to the user and accepts input from the user. In a web application this is the part which receives the HTTP request and returns the HTML response.
2. Business logic - handles data validation, business rules and task-specific behaviour.
3. Data Access logic - communicates with the database by constructing SQL queries and executing them via the relevant API.

These three areas of responsibility follow Robert C. Martin's description of the Single Responsibility Principle (SRP) which he discusses in his article Test Induced Design Damage?

> How do you separate concerns? You separate behaviors that change at different times for different reasons. Things that change together you keep together. Things that change apart you keep apart.
>
> GUIs change at a very different rate, and for very different reasons, than business rules. Database schemas change for very different reasons, and at very different rates than business rules. Keeping these concerns (GUI, business rules, database) separate is good design.

Although it may be possible to take any of the above areas of responsibility and break them down into even smaller components, such as with objects made from different design patterns, it may be unwise to treat each of those objects as a separate layer otherwise you may drown yourself in tiny details and lose sight of the big picture.

Martin Fowler also describes this separation into three layers in his article PresentationDomainDataLayering where he says:

> One of the most common ways to modularize an information-rich program is to separate it into three broad layers: presentation (UI), domain logic (aka business logic), and data access. So you often see web applications divided into a

> web layer that knows about handling HTTP requests and rendering HTML, a business logic layer that contains validations and calculations, and a data access layer that sorts out how to manage persistent data in a database or remote services.

Note here that he refers to the "Business" layer as the "Domain" layer. He also adds in a separate Service layer and a Data Mapper layer, but I consider these to be both useless and unnecessary, so I ignore them. My code works perfectly well without them, so they can't be *that* important.

What do you mean by "logic"?

This may seem like a dumb question to some, but believe it or not there are some numpties out there who cannot identify what is covered by the term "logic" and what is not. They seem to think that everything to do with a computer application is "logic". They see some data in my data access layer which looks like HTML, and they immediately jump up and down like demented children and cry "You have presentation logic in your data access layer! You don't know what you're doing!" They see some data in my presentation layer which looks like SQL, and they immediately jump up and down like demented children and cry "You have data access logic in your presentation layer! You don't know what you're doing!" If you don't believe some of the peculiar opinions out there then take a look at the following:

- Is it really possible to separate business logic from data access logic?
- How can business logic and data access logic be separate if they exist in the same class?
- Information is not Logic just as Data is not Code

Just in case you are one of these numpties, let me explain. Logic is program code, instructions which can be executed. Data is not logic, it is information which may be processed, manipulated or transformed in some way by that program code. Program code is fixed and exists in one place whereas data is variable and can pass through several layers of code. A piece of data may pass through several layers within an application, either from the user interface to the database, or from the database to the user interface, but it is never part of the logic of any of those layers. Data is processed by code, but is never part of the code.

Example 1: The Business layer may contain a piece of data which says "display field X as a dropdown list", but that is not presentation logic which is executed within the Business layer. Presentation logic is that piece of program code which actually constructs the HTML for the dropdown list. All the Business layer is doing is generating an instruction in the form of meta-data which is passed to the Presentation layer where that instruction is actually executed. It is only when that data is processed by the program code in the Presentation layer that the HTML output is actually constructed, and it is the code which produces that output and returns it to the client which is presentation logic.

Example 2: The presentation layer may contain pieces of data which says "Fetch 10 records from the database starting a record number 30, and sort the result on field X", but that is not data access logic. It is only when that data is processed by the program code in the Data Access layer that the SQL query is actually constructed and executed, and it is the code which produces that query and sends it to the database which is data access logic.

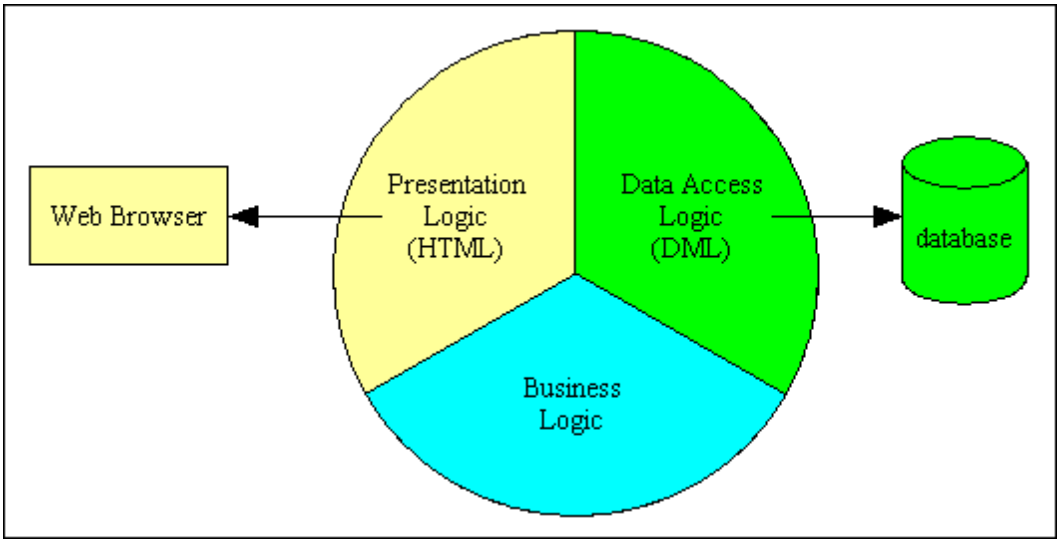
There is a big difference between passing information to another component which can be construed as an instruction and actually executing that instruction. The instruction is merely a piece of meta-data, a piece of information, and the "logic" only exists in that component which executes that instruction in order to produce a result. This difference has been recognised for years as:

- Declarative programming - where you describe *what* the program should accomplish but without defining *when* or *how* those instructions should be implemented.
- Imperative programming - where instructions are actually implemented in the designated sequence.

The 1-Tier Architecture

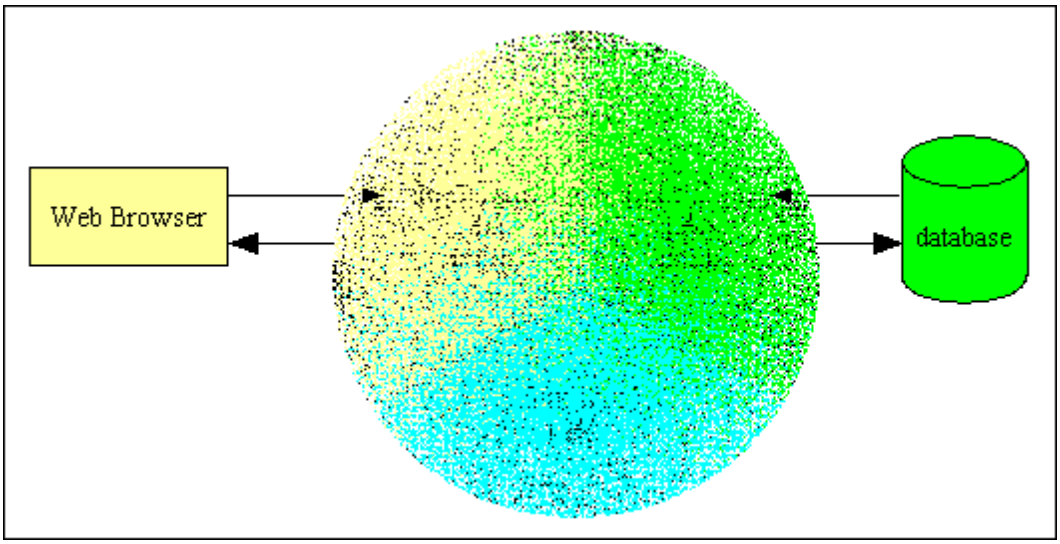
Figure 2 is a simple diagram which shows a 1-Tier application where the Presentation logic, Business logic and Data Access logic are all contained within a single component:

Figure 2 - 1 Tier architecture



Although this diagram apparently makes it easy to identify the different areas of responsibility, in real life the actual program code may be so inter-mingled, inter-twined and spaghetti-like that it would be extremely difficult to locate the boundaries between each area of responsibility. A blurry mess like this is shown in figure 3:

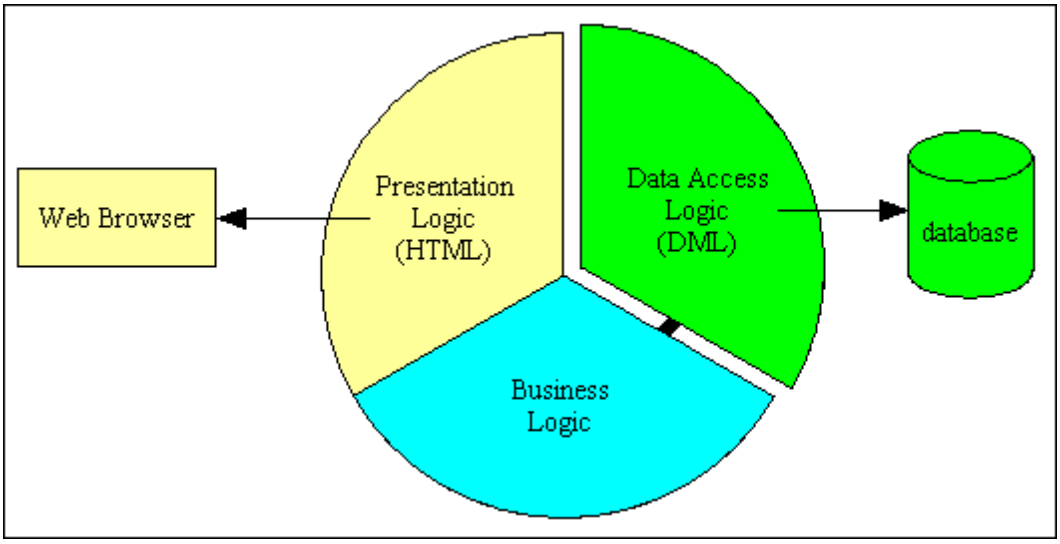
Figure 3 - Typical program with blurry boundaries



The 2-Tier Architecture

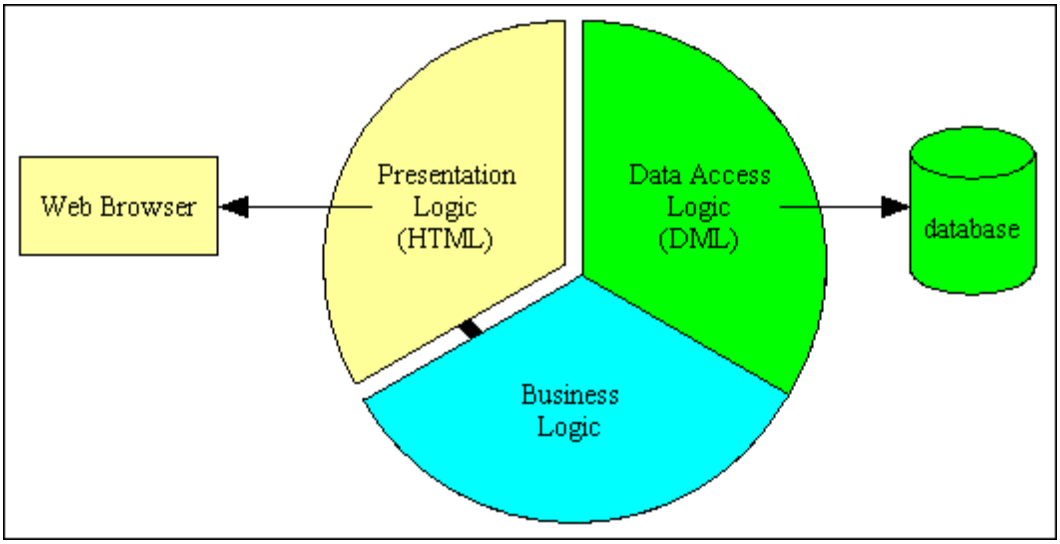
My first exposure to a software architecture which had more than one layer was with a compiled language where all database access was handled by a completely separate component which was provided by the vendor, as shown in figure 4a:

Figure 4a - 2 Tier architecture



Another possible variation is shown in Figure 4b:

Figure 4b - an alternative 2 Tier architecture



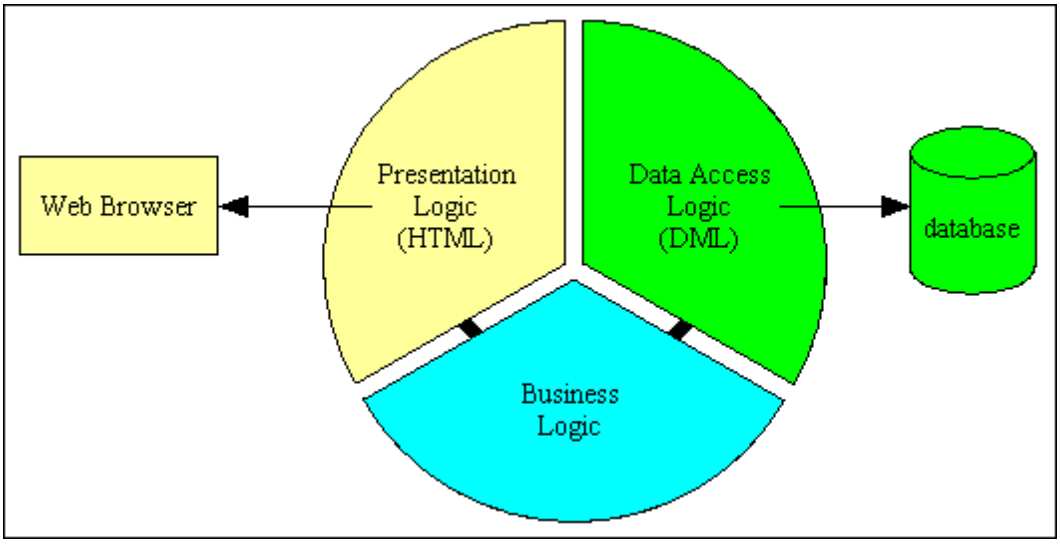
This was a deliberate feature of the language as it enabled an application to be readily switched from one DBMS engine to another simply by loading a different data access component. No other part of the application was allowed to communicate with the database, so this component could be switched without affecting any other part of the application. This made it possible to develop an application using one DBMS, then deploy it with another.

Note that the presentation logic and the business logic are still intermingled.

The 3-Tier Architecture

This is where the code for each area of responsibility can be cleanly split away from the others, as shown in figure 5:

Figure 5 - 3 Tier Architecture



Note here that the presentation layer has no direct communication with the data access layer - it can only talk to the business layer.

Note also that you should not infer from this diagram that the entire application can be built with a single component in each of these three layers. There should several choices as follows:

- There should be a separate component in the Presentation layer for each user transaction.
- There should be a separate component in the Business layer for each business entity (database table).
- There should be a separate component in the Data Access layer for each supported DBMS.

With this structure it is easy to replace the component in one layer with another component without having to make any changes to any component in the other layers.

- You can change the UI component so that you can switch between a variety of different output formats, such as HTML, PDF or CSV.
- You can change the data access component so that you can switch between a variety of database engines, such as MySQL, Oracle or SQL Server.

This structure also provides more reusability as a single component in the Business layer can be shared by several components in the Presentation layer. This means that business logic can be defined in one place yet shared by multiple components.

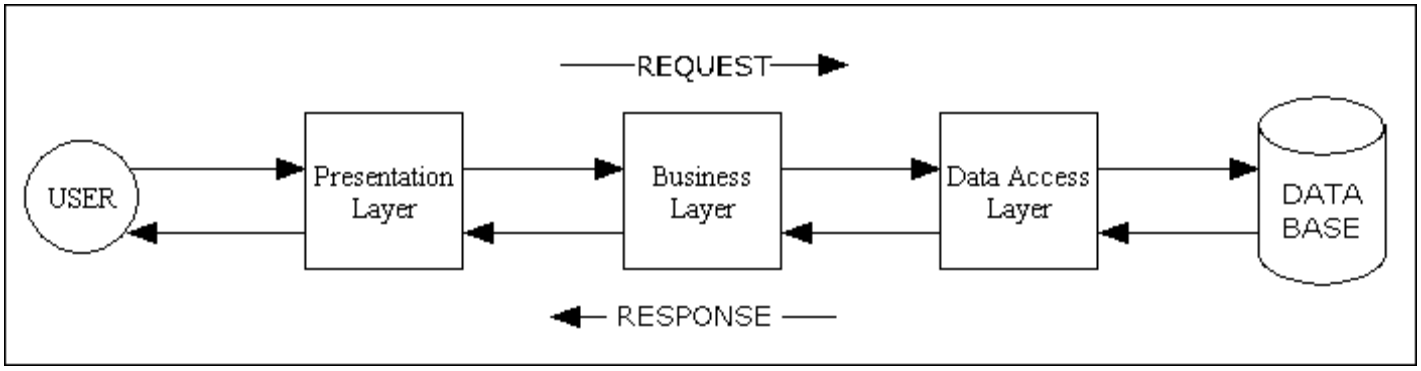
The Rules of the 3 Tier Architecture

It is simply not good enough to split the code for an application into 3 parts and call it "3 Tier" if the code within each tier does not behave in a certain way. There are rules to be followed, but these rules are pretty straightforward.

- The code for each layer must be contained within separate files which can be maintained separately, possibly by separate teams.
- Each layer may only contain code which belongs in that layer. Thus business logic can only reside in the Business layer, presentation logic in the Presentation layer, and data access logic in the Data Access layer.
- The Presentation layer can only receive requests from, and return responses to, an outside agent. This is usually a person, but may be another piece of software.
- The Presentation layer can only send requests to, and receive responses from, the Business layer. It cannot have direct access to either the database or the Data Access layer.
- The Business layer can only receive requests from, and return responses to, the Presentation layer.
- The Business layer can only send requests to, and receive responses from, the Data Access layer. It cannot access the database directly.
- The Data Access layer can only receive requests from, and return responses to, the Business layer. It cannot issue requests to anything other than the DBMS which it supports.
- Each layer should be totally unaware of the inner workings of the other layers. The Business layer, for example, must be database-agnostic and not know or care about the inner workings of the Data Access object. It must also be presentation-agnostic and not know or care how its data will be handled. It should not process its data differently based on what the receiving component will do with that data. The presentation layer may take the data and construct an HTML document, a PDF document, a CSV file, or process it in some other way, but that should be totally irrelevant to the Business layer.

This cycle of requests and their associated responses can be shown in the form of a simple diagram, as shown in figure 6:

Figure 6 - Requests and Responses in the 3 Tier Architecture



If you look carefully at those layers you should see that each one requires different sets of skills:

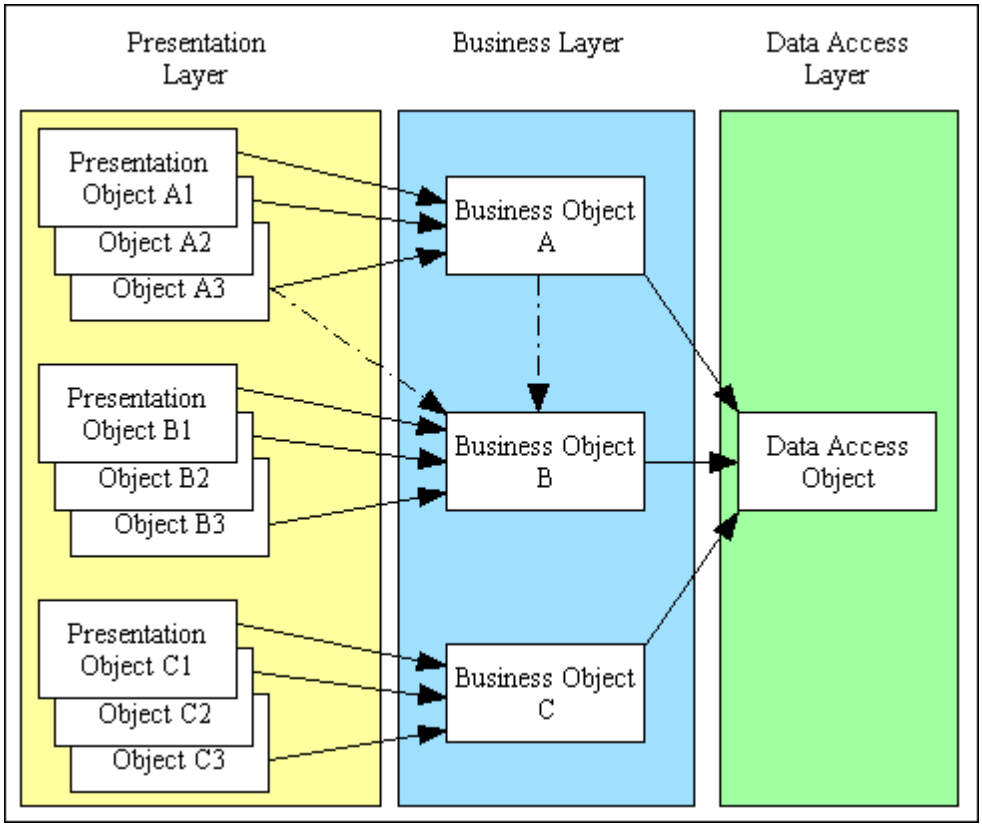
- The Presentation layer requires skills such as HTML, CSS and possibly JavaScript, plus UI design.
- The Business layer requires skills in a programming language so that business rules can be processed by a computer.
- The Data Access layer requires SQL skills in the form of Data Definition Language (DDL) and Data Manipulation Language (DML), plus database design.

Although it is possible for a single person to have all of the above skills, such people are quite rare. In large organisations with large software applications this splitting of an application into separate layers makes it possible for each layer to be developed and maintained by different teams with the relevant specialist skills.

3 Tier Architecture in operation

When an application is developed it is not (or should not be) constructed from a single large component. There are usually lots of small components (sometimes called 'modules', or 'classes' in OOP), each performing a particular function, and the application is the sum of those parts. This allows new functionality to be added without necessarily having any effect on any existing components. The advantage of a layered approach is that a component in a lower layer may be shared by multiple components in a higher layer, as shown in the figure 7:

Figure 7 - 3 Tier Architecture in operation



Here you see that a component in the Presentation layer can communicate with one or more components in the Business layer. A component in the Business layer communicates with the component in the Data Access layer, but may also communicate with other Business layer components. There is usually only one component in the Data Access layer as the application database(s) are usually handled by a single DBMS engine. It is possible to switch to a different DBMS engine simply by changing this single

component. It is also technically possible for different parts of the application to deal with different DBMS engines at the same time, as shown in figure 9.

In my largest application I have 2,000 components (user transactions) in the presentation layer, 250 in the business layer, and 1 in the data access layer. I have heard of some implementations which have a separate Data Access Object (DAO) for each individual table in the database, but more experienced developers can achieve the same functionality with just one. In my own implementation, for example, a single DAO can deal with every table in the database. However, I have a separate class file for each of the major DBMS engines - MySQL, PostgreSQL, Oracle and SQL Server - so I can easily switch from one to another by changing a single entry in my config file.

Note also that the connection to the database is not opened by any component within the Presentation layer. This should only be done within the Data Access layer when instructed to do so by the Business layer, and only the moment before an operation on the database is actually required. This is called the "Just In Time" (JIT) method as against the "Just In Case" (JIC) method. When the Business layer decides that it needs to talk to the database it follows these steps:

- Identify which DBMS is relevant for the database table.
- Instantiate an object from the relevant class file, which is usually a shared singleton.
- Pass a collection of variables to the DBMS object which will be used to construct the relevant query. This allows the query to be constructed according to the needs of that particular DBMS engine. The Oracle and SQL Server databases, for example, do not use OFFSET and LIMIT for pagination.
- Execute the query and wait for the response.
- Deal with the response before returning it as an array. Different DBMS engines, for example, have different methods of dealing with auto-increment columns or sequences, so the code to deal with these differences is contained within the DBMS object and totally invisible to the calling object.

This approach allows an instance of my application to access database tables on more than one server, and even more than one DBMS engine.

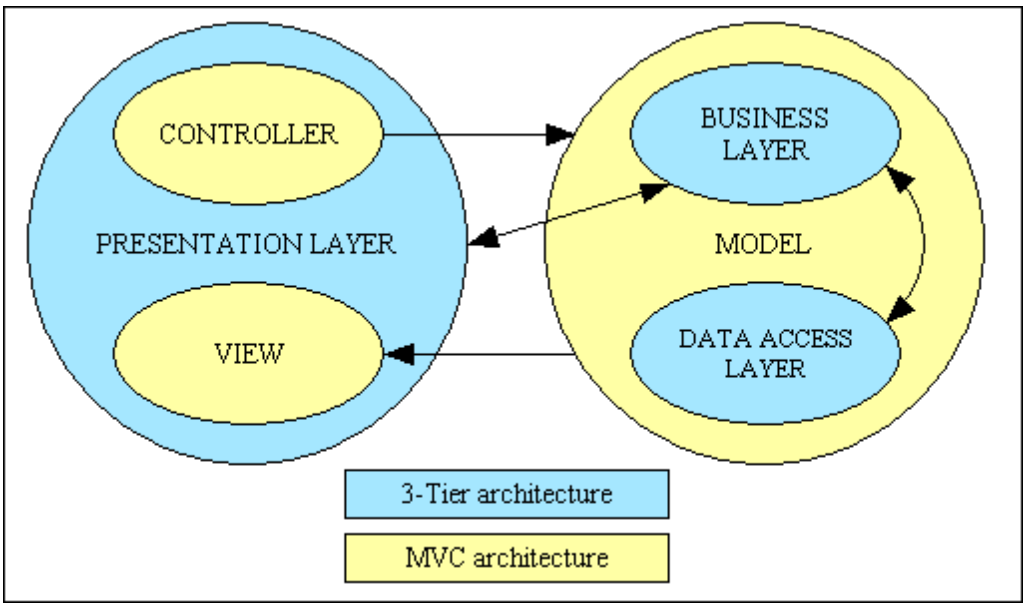
Aren't the MVC and 3-Tier architectures the same thing?

There are some programmers who, upon hearing that an application has been split into 3 areas of responsibility, automatically assume that they have the same responsibilities as the Model-View-Controller (MVC) Design Pattern. This is not the case. While there are similarities there are also some important differences:

- The View and Controller both fit into the Presentation layer.
- Although the Model and Business layers seem to be identical, the MVC pattern does not have a separate component which is dedicated to data access.

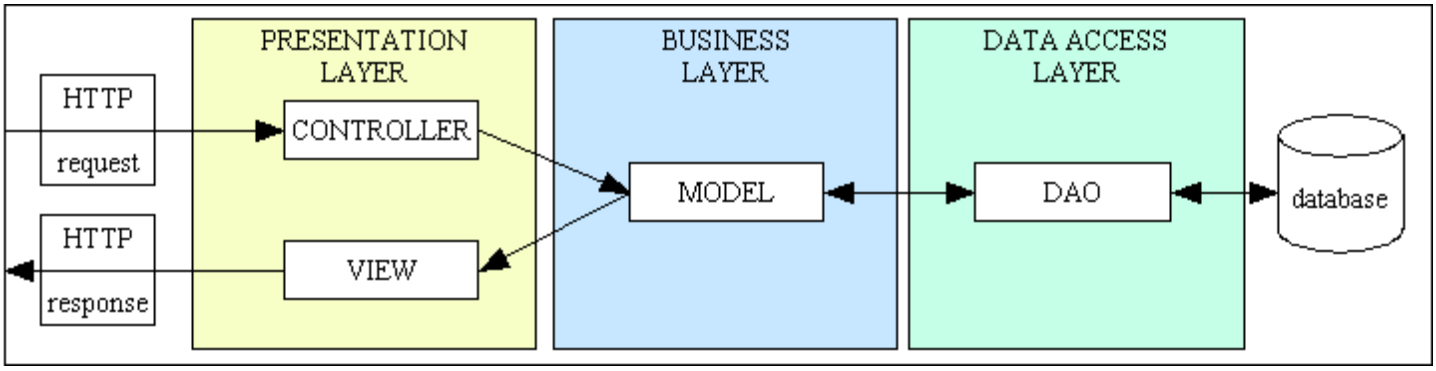
The overlaps and differences are shown in Figure 8 and Figure 8a:

Figure 8 - The MVC and 3-Tier architectures combined



Here is an alternative diagram which shows the same information in a different way:

Figure 8a - MVC plus 3 Tier Architecture



You may think that any implementation of MVC could automatically be regarded as an implementation of 3-Tier, but this is not the case. In every implementation of MVC which I have seen so far there has been one simple but fundamental mistake which makes this impossible, and that is where the database connection is made. In the 3-Tier Architecture **all** communication with the database, and this includes opening a connection, is done within the Data Access layer upon receipt of a request from the Business layer. The Presentation layer does not have **any** communication with the database, it can **only** communicate with it through the Business layer. With all the MVC frameworks I have seen so far the database connection is made within the controller, and the connection object is then passed down to the model which then uses it when necessary. This is a mistake often made by beginners and those who have been poorly educated.

The worst example of this mistake I have ever seen was where a Controller opened up 3 different connections to the same database server instance in order to communicate with 3 different databases in that server. Each of these connections was then passed to a different Model component which may not actually use that connection. An experienced programmer will immediately spot the flaws in this approach, flaws that could have a significant impact on performance:

- Opening up multiple connections to the same server just to access different databases in that server is a waste of resources. A competent programmer will know that once a connection has been established it is possible to switch the identity of the default database, or even to prefix each table name in the SQL query with the relevant database name. It is possible to access multiple tables from multiple databases in a single query, so it is **not** necessary to have a separate connection for each database. All that is required is that the application be aware of both table names *and* their associated database names, and that the DAO be smart enough to remember the current database name so that it can switch to another when necessary. Unfortunately most MVC frameworks assume that all tables are contained within a single database, which makes using multiple databases a bit of a problem.

- Opening up a connection before it is actually required is a waste of resources. This is the "Just In Case" (JIC) method where what should be used is the "Just In Time" (JIT) method.

Another mistake I have found with some implementations is that they think that data validation (sometimes called 'data filtering') should be performed within the controller before it is passed to the model. This does not fit in with the definition of the 3-Tier Architecture which states that such data validation logic, along with business logic and task-specific behaviour, should exist *only* within the business layer or model component. By taking data validation out of the model and putting it in the controller you are effectively creating tight coupling between the controller and the model as that controller cannot be reused with a different model, and tight coupling is something which is supposed to be avoided in order to produce 'better' software. Loose coupling would enable a controller to be shared by multiple model classes instead of being fixed to just one.

When properly implemented, an application which is developed using the 3-Tier Architecture should have all its logic - data validation, business rules and task-specific behaviour - confined to the Business layer. There should be no application logic in the Presentation and Data Access layers, thus allowing either of those two layers to be changed without affecting any application logic.

What are the benefits of the 3-tier architecture?

It is all very well describing something like the 3 Tier Architecture, but nobody is going to make the effort to switch from their current arrangement unless there are benefits to be made, so what exactly are the benefits of this architecture? According to some there are lots of expenses but no benefits at all, as expressed in the following Sitepoint blog:

> It's main function (independence of user interface, business rules and data storage/retrieval) only helps when migrating or extending to another script-language/data engine/platform. But this only happens very very few times in an Application Lifetime.

This is a very narrow-minded view from a person of limited experience. The aim of using the 3-Tier Architecture is **not** just to make it easier **only** when you change your database engine or programming language. It is also useful when you want to either replace your Presentation layer or create an additional Presentation layer. If you have ever worked on a variety of 3-Tier and non-3-Tier applications you would be able to see the differences, and therefore the advantages, for yourself.

The main advantages of the 3 Tier Architecture are often quoted as:

- **Flexibility** - By separating the business logic of an application from its presentation logic, a 3-Tier architecture makes the application much more flexible to changes.
- **Maintainability** - Changes to the components in one layer should have no effect on any others layers. Also, if different layers require different skills (such as HTML/CSS is the presentation layer, PHP/Java in the business layer, SQL in the data access layer) then these can be managed by independent teams with skills in those specific areas.
- **Reusability** - Separating the application into multiple layers makes it easier to implement re-usable components. A single component in the business layer, for example, may be accessed by multiple components in the presentation layer, or even by several different presentation layers (such as desktop and the web) at the same time.
- **Scalability** - A 3-Tier architecture allows distribution of application components across multiple servers thus making the system much more scalable.
- **Reliability** - A 3-Tier architecture, if deployed on multiple servers, makes it easier to increase reliability of a system by implementing multiple levels of redundancy.

Another not-so-obvious benefit which can only come from actual exposure to having developed multiple applications using the 3-Tier Architecture is that it becomes possible to create a framework for building new applications around this architecture. As each of the layers specialises in just one area of the application it is possible to have more reusable components which deal with each of those areas. Such components can either be pre-built and delivered as part of the framework, or generated by the framework itself. This reduces the amount of effort needed to create a new application, and also reduces the amount of effort needed to maintain that application.

If you think that such a framework has not been written yet then you have not heard about Radicore.

The benefits of alternative Data Access layers

A lot of critics of the 3-Tier Architecture seem to think that the ability to switch from one DBMS to another is not worth the effort as it rarely happens in real life. While it is true that, once developed, an application will rarely be switched to a different DBMS, a lot of organisations are now wishing that they had that ability when they compare the costs of their proprietary databases with the modern and far cheaper open source equivalents.

Switching to a different DBMS **after** an application has been built is not the only benefit when you consider that not everyone works on a single legacy application for a single organisation. There are software houses who develop and maintain a variety of applications for a variety of customers, so speed of development, cost of development, and providing their customers with more choices is what differentiates them from the competition and is likely to be a selling point instead of a rarely-used option. Consider the following:

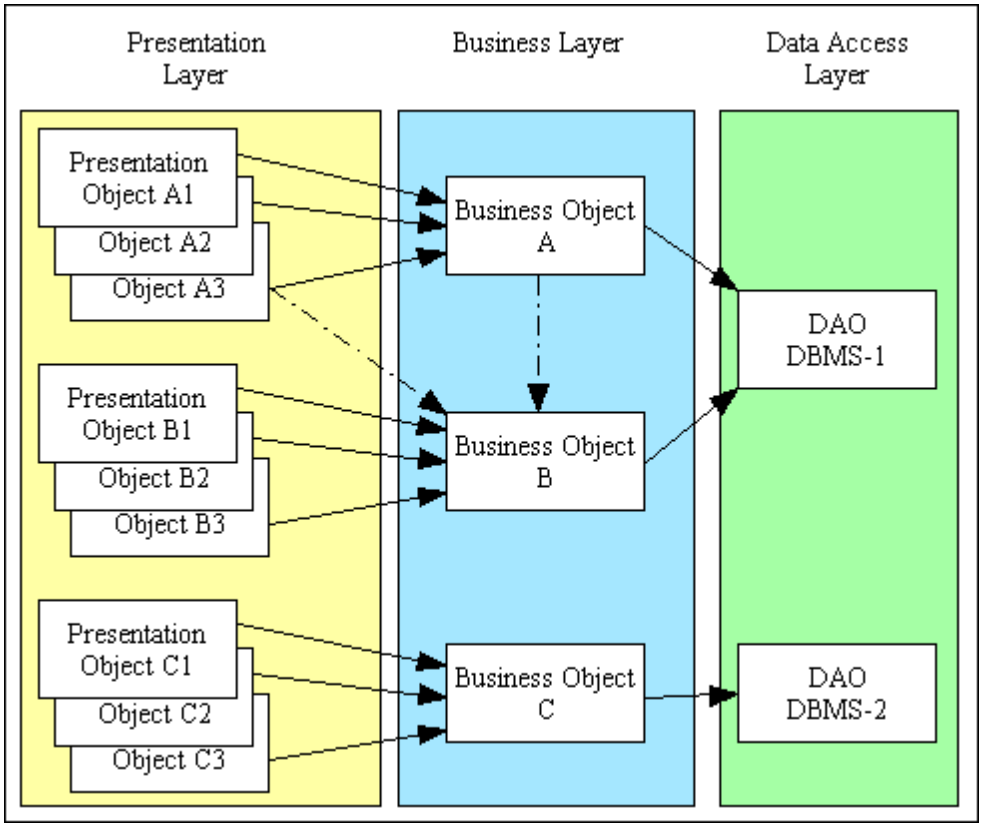
- Suppose you develop an application which you want to sell as a package to lots of different customers? Do you want to restrict your potential customers to a DBMS of **your** choice, or one of **their** choice? By having all the DBMS logic in its own component in its own layer you can deliver the same application code to everybody and let them decide on what DBMS to use at installation time.

- Suppose that, instead of developing actual end-user applications, you develop a framework for building end-user applications? Do you want to restrict the potential users of this framework to a DBMS of **your** choice, or one of **their** choice?

By giving your customers the ability to easily switch from one DBMS to another without enormous expense and effort you will be pleasing your customers and displeasing your competitors.

The ability to switch from one DBMS to another does not, or should not, restrict you to just one DBMS at a time. If each component in the Business layer is responsible for creating and communicating with a component in the Data Access layer it should be possible to allow more than one component to exist at the same time, as shown in figure 9:

Figure 9 - Using multiple objects in the Data Access layer



Here you can see that instead of just "alternative" components in the Data Access layer you actually have the option of "additional" components. This will allow, for example, part of your application to access a MySQL database while another part accesses a PostgreSQL/Oracle/SQL Server database.

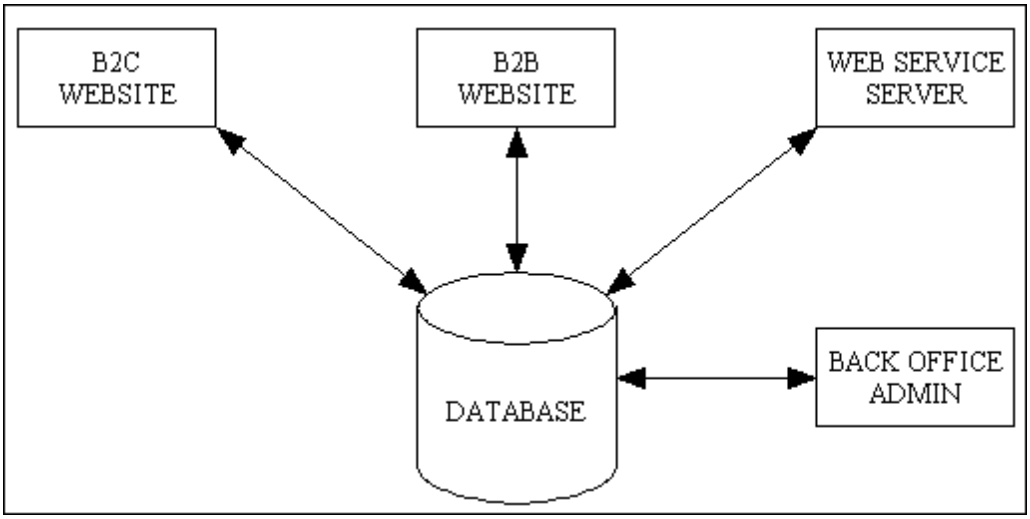
The benefits of alternative Presentation layers

A lot of critics of the 3-Tier Architecture seem to think that the ability to switch from one Presentation layer to another is something else which has very little value in the real world, but yet again they are being narrow-minded and blinkered in their viewpoint. I first encountered the 3-Tier Architecture when using a compiled language which produced applications for the desktop. As soon as the internet revolution got into full swing and more and more organisations wanted a web site, the authors of this language added in the ability to generate a new presentation layer based around the HTTP protocol and HTML forms. This new Presentation layer still accessed the existing Business and Data Access layers, so it did not involve a full application rewrite, just the development of a new set of components for the Presentation layer. This new layer was not *instead of* the old layer, it was *in addition to* the old layer, which made it possible for the original desktop application to be used by the in-house staff while internet visitors used the new HTML layer. Developers who used this language liked the idea that they could build a web interface into their existing application and make use of existing components instead of having to rewrite the whole application from scratch.

The ability to switch from one Presentation layer to another does not, or should not, restrict you to just one Presentation layer at a time. Each Presentation layer has its own set of components and is activated in a different way by a different set of users, but they all then communicate with the same set of components in the Business and Data Access layers. It should then be possible to allow more than one Presentation layer to be used at the same time. Thus instead of just an "alternative" Presentation layer you actually have the option of "additional" Presentation layers. This will allow one set of users to access the application through one Presentation layer while other sets of users access it through different Presentation layers which have been tailored just for them.

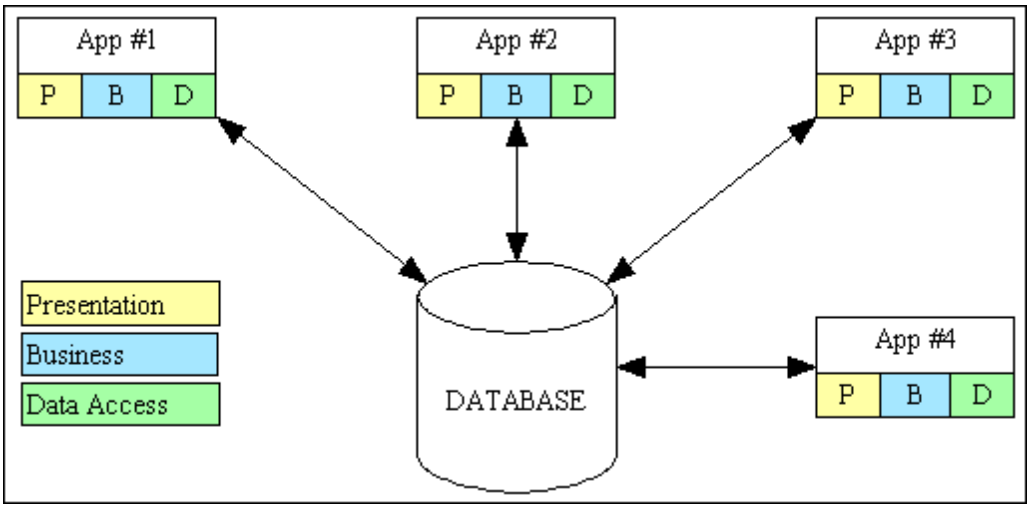
This choice between two Presentation layers, one for in-house staff and another for internet visitors, has been extended in recent years to cover additional sets of users. I have personally worked on several e-commerce applications which, as well as a Business-to-Consumer (B2C) site they also have a Business-to-Business (B2B) site - sometimes known as a Supplier Portal - and increasingly an additional site for dealing with web services. I have also helped an organisation go multi-national with copies of their UK website ported to different servers in different languages, all being served from the same back-end application and database. This then produces an application which has several Presentation layers or "windows", as shown in figure 10:

Figure 10 - A typical application with multiple "windows"



Just suppose each of these "windows" was treated as a separate application instead of a small part of a much larger application? None of these applications would share any component in any of the other applications, which would result in a duplication of effort. Even though each of these "windows" would have its own presentation logic, it would also have its own business logic and data access logic. This would result in the situation shown in figure 11:

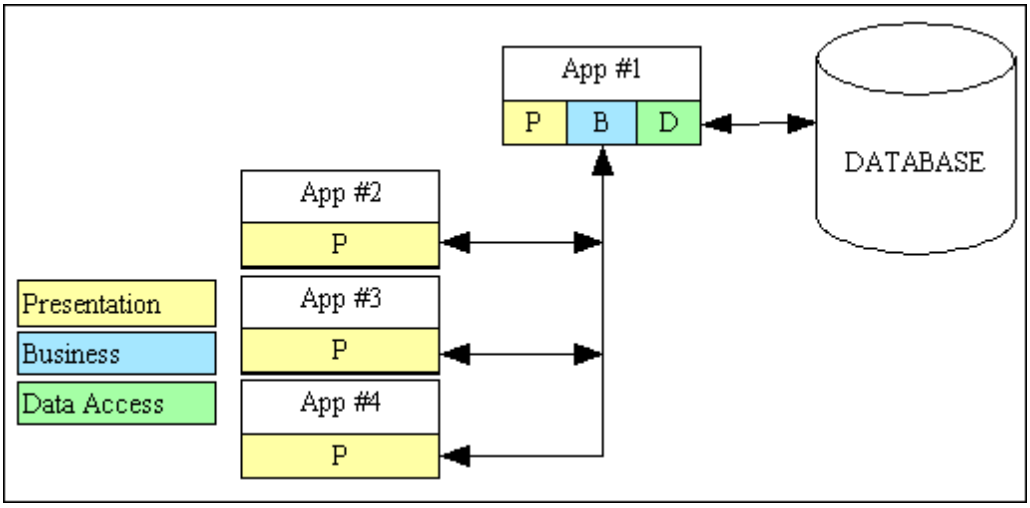
Figure 11 - Each "window" has its own layers



If you wanted each of these "windows" to enforce the same business rules you would have to maintain the program code in each of the different business layers, which would be a costly duplication of effort as well as a potential problem area should the code in one "window" not be kept in sync with the others.

Just suppose the first (or primary) application had been developed using the 3-Tier Architecture with its reusable Business layer and Data Access layer components? All the additional (or secondary) "windows" would be much smaller as they would not require their own set of business and data access logic, they would be able to share the logic which was written for the first application. This arrangement is shown in figure 12:

Figure 12 - Multiple Presentation layers sharing a single Business layer and Data Access layer



Here you can see that only one application - the back end management application - has all three layers of code while all the other applications are nothing more than simple "windows" which consist of a separate presentation layer which connects to the shared business layer (and through that the shared data access layer).

As the code to enforce the business rules is maintained in a single place then you also eliminate the problem of synchronising those rules in multiple places. You can make a change to a single component in the Business layer and have that change immediately picked up (or "inherited" using the parlance of Objected Oriented programmers) by all the various "windows".

As each of these additional "windows" is able to reuse a significant amount of existing code, the observant among you will immediately notice that this should also make them quicker, easier and cheaper to develop. How much cheaper depends on how you build your front ends. For example, if you are a follower of the Model-View-Controller design pattern you should see that the existing Business layer already provides the Model, so you have nothing to design and develop for each front end except the Controller and the View. The skills required here are little more than HTML, CSS and JavaScript, while the heavy lifting - the database design, the processing of business rules and the generation of complex SQL queries - can be left to the big boys in the back office.

It is also possible for an object in the business layer, a domain/business object, to be accessed by multiple objects in the presentation layer, one for each different task, where the raw data which is extracted from the business object can be presented to the user in the format which is required by that particular task. Thus the presentation layer could take the same raw data and format it as HTML in one task, CSV in a second, and PDF in a third. In all these cases the business object itself does not do any formatting, it simply returns raw data to the presentation layer, and it is the presentation layer which does the formatting.

The benefits of multiple Presentation components

Some people seem to think that an object in the business layer can only ever have a single corresponding object in the presentation layer, but as I have never seen such a rule, nor have I seen any justification for such a rule, I choose to follow my own instincts and incorporate practical solutions to real-world problems.

My first programming language was COBOL, and it was common practice in those days (the late 1970s and early 1980s) to create a single program for a business entity which handled all the operations which could be performed on that entity as well as all the screens required by each of those operations. This resulted in a small number of large and complex programs. As time went by the number of entities grew larger and the user requirements became more sophisticated. If an application contained different programs for different areas of the business, such as Customers, Products, Orders, Inventory and Shipments, it was usually a requirement to have some sort of Access Control List (ACL) so that each user could only access those programs which were necessary for he/she to complete their daily tasks. The ACL simply linked a user to a program. This meant that a user, if given permission to access a particular program, could then access all the operations within that program.

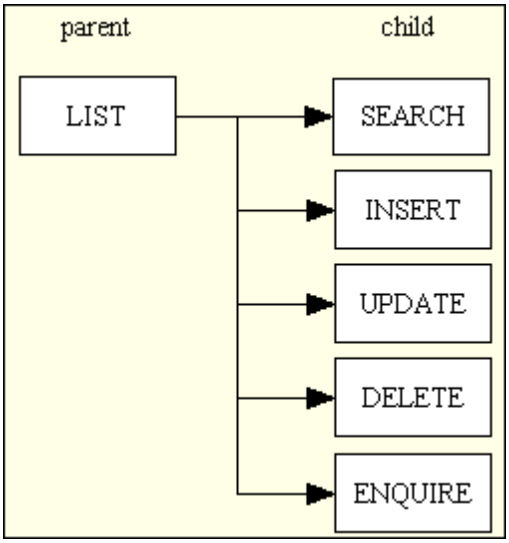
Then the requirement changed so that a user was not just given permission to access the program with all its operations but only some of those operations. For example, a manager might be able to create, update and delete entries from the Product database, but his juniors could be restricted to the search, list and enquire operations. As well as requiring a modification to the ACL to have three parts (user, program and operation) instead of two (user and program) it also required additional code within each program so that when switching from one mode to another, such as from "enquire" to "update", the ACL had to be checked to ensure that the switch was allowed, and to output an error message if it was not. Some of my junior team members struggled with this extra complexity, and sometimes the extra code would push the program over the memory limit and it wouldn't even compile. Something clearly had to change.

It was obvious to me that if a program which performs multiple operations was too large to be compiled then one or more of those operations had to be separated out, so I decided to go the whole hog and separate them all so that each operation was in its own program. This resulted in a large number of small and simple programs instead of a small number of large and complex programs.

At first the members of my team thought that by increasing the number of programs it would also increase the timescales, but the opposite was true. Each program was smaller and less complex and the overall timescales were reduced. The code to check the ACL was removed from each program, where it had been duplicated, to a single place in the framework which I was in the process of developing. This was a Menu and Security system which held details of the users who were allowed to access the system, a list of all the programs (user transactions) which were available within the system, and an access profile which identified those transactions which a user was allowed to access. It also had a table of menus so that the list of available transactions could be displayed to the user in a hierarchy of menu pages so that the user could quickly identify and execute the one that he wanted. As the menu pages were dynamic (read from the database) instead of static (hard coded) it also meant that the framework could filter this list to remove the reference to any transaction to which the user had not been granted access. This topic is discussed in greater detail in Component Design - Large and Complex vs. Small and Simple.

For an example of what this means take a look at Figure 13 below:

Figure 13 - A typical Family of Forms



Note: each of the boxes in the above diagram is a clickable link.

While in my early COBOL days it was common practice for these six operations to be coded within the same program I later discovered that it was much better to have a separate program for each operation. Each of these "programs" is actually a separate component in the presentation layer which all access the same component in the business layer. This means that I can add, change or remove a component from the presentation layer without having to change any code in the business layer. For example, I could add a new component using the OUTPUT1 pattern which produces CSV output instead of HTML without having to change anything in the business layer as I would be able to share an existing business layer component. This is possible because the business component does nothing but return raw data to the presentation layer, and it is the presentation layer which is responsible for transforming that raw data into the format required by the user. There is no limit to the number of components in the presentation layer which can access the same component in the business layer.

Why do the Front End and Back End require different Presentation layers?

This is a question I am often asked by inexperienced people, and the reasons, which are detailed in Web Site vs Web Application, can be summarised as:

- They are used by different people
- They serve a different purpose
- They have a different "look and feel"

The screens in the front end need to attract visitors, just like a showroom or a high street store. They are free-form and often built using complex HTML, CSS, JavaScript and lots of sexy graphics. They may also contain code to deal with Search Engine Optimisation (SEO), analytics, affiliate marketing, advertising banners and other marketing stuff.

The screens in the back end are a lot simpler as all they need do is allow members of staff to do their jobs as quickly and efficiently as possible, and therefore need to be functional rather than sexy. They don't need complex HTML, CSS or JavaScript, and certainly don't need sexy graphics, SEO or other marketing stuff. As they deal with database data which is organised into rows and columns they can be boiled down into two common screen structures - the LIST view which shows multiple rows in a horizontal arrangement and the DETAIL view which shows a single row in a vertical arrangement. This means that the screens in the back end can be built more easily from standard templates whereas the screens in the front end are usually hand crafted one by one. There are usually some additional screens in the back end which might need complex SQL queries so that the data can be extracted and presented in a variety of management reports.

There are also more screens in the back end than in the front end (the ratio can be as much as 100:1), and not every function that is available in the back end will be available in the front end while anything which can be done in the front end should also be possible in the back end. For example, in e-commerce applications the front-end website is used by customers to enter their orders, but there should always be an order entry function available in the back office.

Because of these significant differences between the front end and the back end is should be easy to see that:

- They require different development skills.
- They may be built in different languages.
- If built in the same language they may use different frameworks.

In my opinion it would be extremely unwise to see the back end application as an extension to the front end website, or the front end as an extension to the back end. The back end administrative application gives its users, who are members of staff (and possibly management), complete access to every aspect of the company's business. The front end website, on the other hand, gives its users, who are casual visitors, extremely limited access.

Front End Thin Clients and Back End Server

You should have noticed in the preceding paragraphs that I have referred to the first or "primary application" and a series of one or more "secondary windows". When developing a website for a new business the most important question to be asked is "What is the primary application and what are the secondary windows?" This is where a large number of developers make a simple but costly mistake by choosing the front-end website as the primary application and totally ignoring the need for a back-end administrative application.

Those of you who have been involved in the development of e-Commerce applications for various organisations should immediately recognise the folly in this decision. Just as an organisation would not be foolish enough to open up a bricks-and-mortar store in the high street without having all the necessary infrastructure in place, an organisation should not attempt to open up an internet store without having all the necessary software infrastructure in place. If you don't know what I mean by infrastructure then you should go to the back of the class and hang your head in shame. By "software infrastructure" I mean the following:

- A properly designed and normalised database. There must be a place for every piece of relevant data, and every piece of data must be in the right place.
- A back-end administrative application which allows members of staff to view and maintain every piece of data in the database, and to monitor the performance of the organisation's business.

The needs of the front-end website, the electronic store, are quite simple:

1. Display products.
2. Accept sales.

The needs of the back-end application - sometimes referred to as Order Processing, Order Fulfilment, Logistics, Supply Chain Management or Enterprise Resource Planning (ERP) - cover every aspect of the organisation's business:

1. Maintain product data - products, product categories (which may have hierarchies), product features (availability, applicability and compatibility), product prices (standard prices, contract prices), surcharges (such as as sales tax and shipping), discounts (either fixed value or a percentage).
2. Maintain party details - customers and suppliers, classifications, roles, contact mechanisms (postal addresses, email addresses, telephone numbers).
3. Contract prices and/or discounts, either for specific parties or for party categories.
4. Customer Relationship Management (CRM) - emails received, emails sent, notes on conversations.
5. Content Management System (CMS) - maintaining text for static pages.
6. Maintain order details - sales orders, order adjustments, item adjustments, purchase orders, non-conformance reports, transfer orders (for inventory).
7. Invoicing - invoices, credit notes, PDF generation, financial reports.
8. Shipments - customer shipments, customer returns, supplier shipments, supplier returns, transfers (in and out), packages, package contents, returned material authorisations (RMAs from customers) and non-conformance reports (NCRs for suppliers).
9. Inventory - facilities, containers, items, receipts, pick lists, issuances, stock checks, variances, lot numbers, reorder guidelines.
10. Management reports.

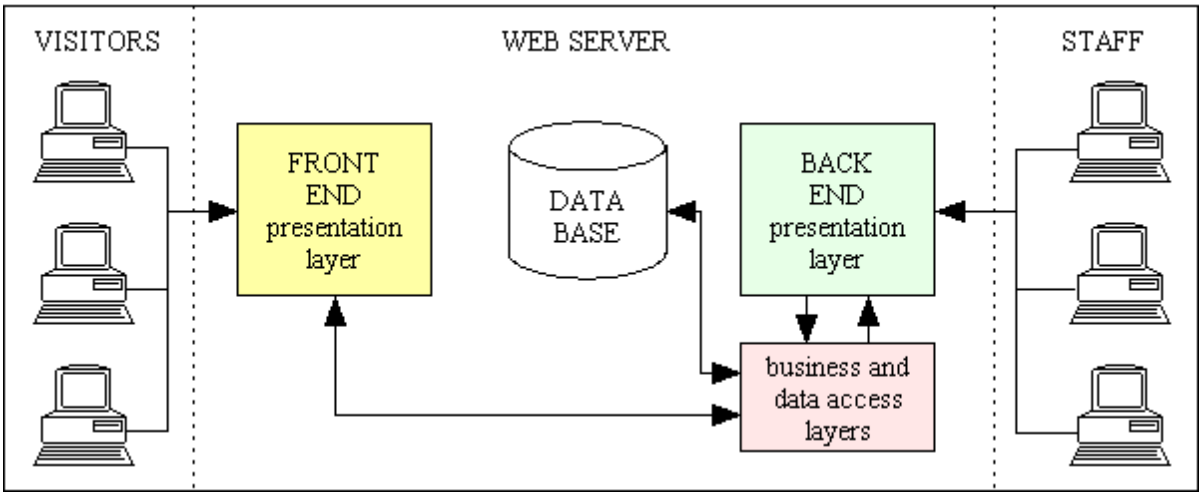
Just as a high street store cannot exist without the infrastructure to support it, neither can an electronic store (the website) exist without its administrative application. Just as in the world of bricks and mortar the infrastructure is built up first and the stores in the high street are opened up last, so should it be in the world of electronic stores. Just as in the physical world the high street stores use the procedures and processes laid down by the management team in head office, in the electronic world the front-end

websites should use the procedures and processes built into the back-end management application. Just as in the physical world each high street store is merely a small representation of the business as a whole, in the electronic world each front-end website is nothing more than a small window into the much larger back-end management application - it needs nothing more than a small presentation layer which connects into the back-end business layer. Just as in the real world any high street store can receive a make-over without having any effect on the infrastructure or the head office, in the electronic world a website's presentation layer can receive a make-over without having to rewrite any of the business logic in the back-end administrative application.

So, just as in the world of bricks and mortar the supporting infrastructure is built first and any customer-accessible stores are built last, in the electronic world the back-end administrative application should be built first and any customer-facing websites built last.

The centre of the software application is always the data which is stored within the database, with members of staff and visitors each having their different "windows" into that data. The business rules are constructed by the administrators and held within the larger back-end administrative application, yet shared with any number of smaller front-end websites. This arrangement is shown in figure 13:

Figure 14 - A small Front End sharing the components in a larger Back End application



The primary purpose of a front end website is presenting the organisation to the outside world in order to attract customers. The primary purpose of the back end application is the administration of the organisation's data and the enforcement of business rules. They both share the same pool of data, they both share the same set of business rules, and it is only in the way that the data is presented to their respected sets of users, and the functions which they are able to perform, which is different. You should see that the terms "presentation", "business rules" and "data" are a perfect match for the different layers in the 3 Tier Architecture, so by using this architecture to build your software you will be able to have any number of unique presentation layers which all share a single business and data access layer.

When I say that the back-end application is larger than the front-end websites I don't mean slightly larger, I mean orders of magnitude larger. For example, I have built a back-end e-commerce application as a package which is used by several different companies selling different goods through their own bespoke front-end websites. They each use a copy of the same back-end software with their own copy of the database. The average number of screens in these front-ends is about 20, so when you compare this with my back-end application which has 2,000 screens you will see that each front-end is only 1% the size of the back-end. To put in another way, my back-end application is 100 times larger than any front-end. Because the front-ends are so small (they require nothing more than a simple presentation layer which can communicate with the existing business and data access layers) it is a relatively quick, and therefore inexpensive, process to build a new bespoke front-end for a new client, yet they have immediate access to a large and powerful back-end application.

Because the front-end website calls upon the services provided by the back-end application the two can be said to exist in a client-server relationship with the front-end being the "client" and the back-end being the "server".

Example code - Front End to Back End (direct access)

Describing how easy it is for the front end to connect to the back end is not very convincing without some example code. Below are some examples of functions which the front end can use to retrieve product data. These use the **singleton::getInstance** method to create an instance of the 'product' class (which exists in the Business layer of the back end application), load in optional data where necessary, then activate the **getData** method to retrieve data from the database. Some of the more complicated options which are available have been left out for the sake of simplicity.

Note that in keeping with the rules of the 3-Tier Architecture the Presentation layer does not connect to the database and pass this connection to the Business layer object. It is up to the Business layer object to make the connection when it is necessary.

The **get_product_array** function will retrieve those records which match the `$where` criteria and will be sorted by the `$orderby` value. Pagination is controlled by the values for `$pageno` and `$rows_per_page`.

```
function get_product_array ($where=null, $orderby=null, $pageno=1, $rows_per_page=12)
// get array of multiple products
{
    if (empty($where)) return false;

    $dbobject = RDCsingleton::getInstance('product');
    $dbobject->setOrderBy($orderby);
    $dbobject->setPageNo($pageno);
    $dbobject->setRowsPerPage($rows_per_page);
    $rowdata = $dbobject->getData($where);

    $numrows = $dbobject->numrows;
    $lastpage = $dbobject->lastpage;

    return array($rowdata, $numrows, $lastpage);

} // get_product_array
```

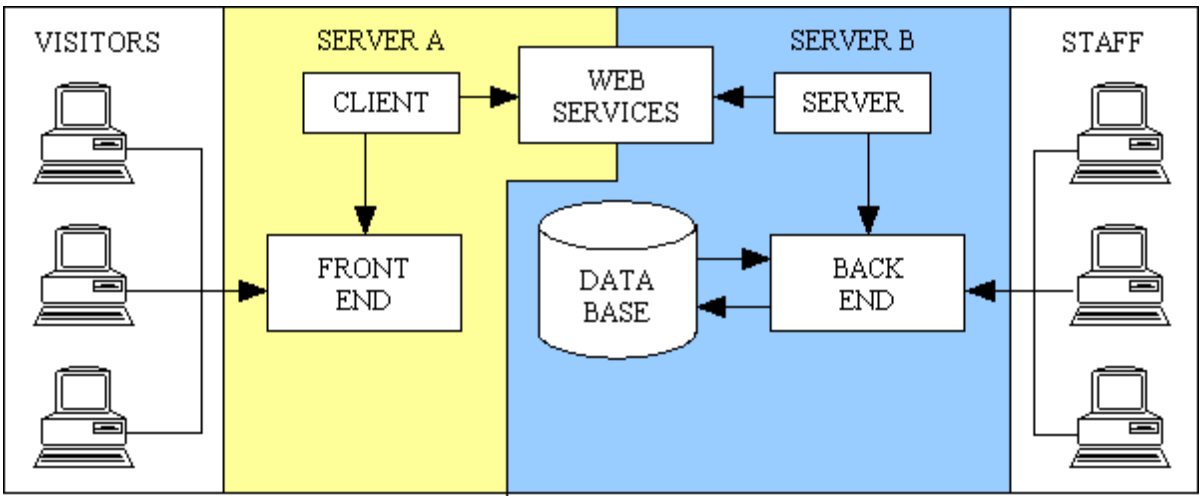
The **get_product_data** function will retrieve the details for a single product which is identified by its primary key.

```
function get_product_data ($product_id=null)
// get details for a single product
{
    if (empty(product_id)) return false;

    $where = "product_id='$product_id'";

    $dbobject = RDCsingleton::getInstance('product');
    $data = $dbobject->getData($where);
    if (!empty($data)) {
        $data = $data[0]; // return 1st row only
    } // if

    return $data;

} // get_product_data
```

Web Service Clients and Web Service Server

Experienced software developers may spot a small limitation within the above arrangement - in order for each Presentation layer to communicate directly with the Business layer both layers must be written in the same language and reside on the same server. It is possible for different Presentation layers to reside on different servers, but only if they have their own copies of the code for the Business layer and Data Access layers. The same experienced software developers should also know of a different approach - web services. This means that instead of having to communicate directly with the back-end application the front-end simply sends a request over the internet which is received, processed, and the response returned over the internet. Thus the front-end website acts as a web service client while the back-end application acts as a web service server, as shown in figure 15:

Figure 15 - Linking the front and back ends with Web Services



While it is possible for the web service client and server to exist on the same physical machine and be written in the same language (I do all my development and testing on a single PC, for example), it is not uncommon for the client and server to be written in different languages and be located on servers which are miles apart. This opens up a world of opportunities.

Example code - Front End to Back End (using web services)

In the following examples I have modified the previous functions so that I can switch from direct access to web service access simply by setting an option in the CONFIG file. These use the **get_TRANSIX_client** function to obtain an instance of the web service client which sends the request over the internet to the web service server which has been built into the back end application. This then uses a copy of the function which has direct access to the database. Even though the response is returned in XML format it is automatically converted into a PHP array so the front end Presentation layer does not have to know how the data was obtained. It simply calls a function to get some data, and how that data is obtained is irrelevant.

Note that it is physically impossible to send a database connection over the internet, so the fact that the Presentation layer never creates one in the first place means that I don't have a connection which is wasted.

This function retrieves multiple records with pagination options. Note that the web service client uses the **getMany()** method for this purpose.

```
function get_product_array ($where=null, $orderby=null, $pageno=1, $rows_per_page=12)
// get array of multiple products
{
    if (empty($where)) return false;

    if (defined('USE_WEB_SERVICES')) {
        require_once('transix_client_api.inc');
        $dbobject = get_TRANSIX_client('Product');
        $args['where']     = $where;
        $args['orderby']   = $orderby;
        $args['page_size'] = $rows_per_page;
        $args['page_no']   = $pageno;
        $rowdata = $dbobject->getMany($args);
    } else {
        $dbobject = RDCsingleton::getInstance('product');
        $dbobject->setOrderBy($orderby);
        $dbobject->setPageNo($pageno);
        $dbobject->setRowsPerPage($rows_per_page);
        $rowdata = $dbobject->getData($where);
    } // if

    $numrows = $dbobject->numrows;
    $lastpage = $dbobject->lastpage;

    return array($rowdata, $numrows, $lastpage);

} // get_product_array
```

This function retrieves a single record which is identified by its primary key. Note that the web service client uses the **get()** method for this purpose as it is only expected to provide a single row from the database.

```
function get_product_data ($product_id=null)
// get details for a single product
{
    if (empty($product_id)) return false;

    $where = "product_id='$product_id'";

    if (defined('USE_WEB_SERVICES')) {
        require_once('transix_client_api.inc');
        $dbobject = get_TRANSIX_client('Product');
        $args['where'] = $where;
        $data = $dbobject->get($args);
    } else {
        $dbobject = RDCsingleton::getInstance('product');
        $data = $dbobject->getData($where);
        if (!empty($data)) {
            $data = $data[0]; // return 1st row only
        } // if
    } // if

    return $data;

} // get_product_data
```

Example code - business rules in the Business layer

One common question I am asked by novice programmers is "How can you put the code for your business rules in the Business layer component?" A lot depends on how the component is actually built, but the RADICORE framework makes this easy by providing components which have empty compartments into which you can insert your code. Each of these compartments will be executed at a predetermined point in the processing sequence, such as pre-insert, post-insert, pre-update, post-update, so when you put code into one of these compartments you know exactly when it will get processed.

For example, suppose there is a business rule which states "when an order's status is changed this must be recorded in the status history table". This can be achieved with the following code in the _cm_post_updateRecord() method of the ORDER_HEADER class:

```
function _cm_post_updateRecord ($rowdata, $old_data)
// perform custom processing after database record is updated.
{
    if ($rowdata['order_status_type_id'] != $old_data['order_status_type_id']) {
        $dbobject = RDCsingleton::getInstance('order_status_hist');
        $data['order_id']             = $rowdata['order_id'];
        $data['order_status_type_id'] = $rowdata['order_status_type_id'];
        $data = $dbobject->insertRecord($data);
        if ($dbobject->errors) {
            $this->errors = array_merge($this->errors, $dbobject->errors);
            return $rowdata;
        } // if
    } // if

    return $rowdata;

} // _cm_post_updateRecord
```

Note that the function which calls the updateRecord() method on the ORDER_HEADER object is totally oblivious to this business rule, so does not require extra code to carry it out. This mechanism can be extended so that the ORDER_STATUS_HIST component, when inserting a record with a particular status value, can automatically send out an email to the customer regarding that change in status. Again this logic is hidden from the ORDER_HEADER object, or any other object which calls the insertRecord() method on the ORDER_STATUS_HIST component.

By putting the code which carries out a business rule into the right place in the right object it is possible to guarantee that the rule will be carried out by all users of that object. By putting the code into the Business layer which is shared by multiple Presentation layers you can guarantee that each of those Presentation layers will share the same business logic without even being aware that any logic exists. So when a Presentation component issues the instruction "update order status" it is unaware that a record is written to the history table or that an email is sent out. In this way each front end Presentation layer can be kept as lean and lightweight as possible as all the heavy lifting is carried out by Business layer components in the back end.

Conclusion

I hope that I have managed to convince you that the 3-Tier Architecture is not just an academic exercise with huge costs and limited benefits. Once you have made the first step then other possibilities come within easy reach:

- It has a much cleaner separation of responsibilities than the Model-View-Controller design pattern in that business logic and data access logic are maintained in totally separate components.
- It gives you the option of using different teams with different skills to develop each of the different layers.
- A single business layer with all that complicated business logic can be used with different data access and presentation layers.
- It provides the ability for multiple data access objects at the same time, not just swappable objects.
- It provides the ability for multiple presentation layers at the same time, not just swappable presentation layers.
- It gives you the option of creating front-end websites which consist of nothing more than an additional presentation layer which hook into the business and data access layers created for the back-end administrative application.
- Each front-end then becomes a simple "thin client" to the back-end "thick server".
- The front-end websites, now being much smaller, can be developed faster and cheaper than before.
- A client-server architecture with direct communication can easily be upgraded to use web services so that the clients can be hosted on different machines and even written in different languages.

When implemented correctly the 3-Tier Architecture can provide a whole series of benefits which would otherwise be very expensive to implement. I would ask you to try it and see, but your current architecture is probably so outdated and badly written that it you can do no more than dream about it. What bad luck!

References

Definitions:

- Three-Tier Architecture from the Linux Journal
- Three-tier architecture from wikipedia.org
- PresentationDomainDataLayering by Martin Fowler
- 3-Tier Architecture: A Complete Overview from jinfonet.com
- What does 3-Tier architecture mean? from techopedia.com
- How to Organize Application Code With 3-Tier Architecture? By Pavel Kukhnavets
- 3-Tier Architecture from ADL Data Systems
- Client/Server and the N-Tier Model of Distributed Computing from n-tier.com

Articles:

- The 3-Tier Architecture - is it hardware or software?
- The 3-Tier Architecture in the Radicore framework
- The 3-Tier Architecture (Design Pattern)
- The Model-View-Controller (MVC) Design Pattern for PHP
- What are the benefits of the 3-tier architecture?
- Aren't the MVC and 3-Tier architectures the same thing?
- Isn't every web application automatically 3-tier?
- Is it really possible to separate business logic from data access logic?
- Web Site vs Web Application
- Why you should build your web application back-to-front

If you want to read some criticisms of my approach then please take a look at the following:

- Your system is not 3 Tier
- You don't understand what '3 tier' means

Amendment History

11 Mar 2021	Added The benefits of multiple Presentation components
28 Apr 2020	Added Definitions to provide a list of definitions taken from other sources.
