SILHOUETTE SOLUTIONS

Data and Business Systems

Dr J A Briginshaw

21 May 2008

Silhouette Solutions Limited

www.silhouettesolutions.com

j.briginshaw@silhouettesolutions.com

Abstract

We discuss the problems businesses typically encounter when employing multiple business software applications as part of their IT infrastructure. Issues preventing effective sharing of data between applications are highlighted and techniques for building a 'single view' of business data are discussed. We talk about appropriate placement of business logic, and we briefly cover some technical issues with interfacing object-oriented programming languages to relational databases, and how to make interfaces between different business systems more resilient to change.

Introduction

Flow of data is the lifeblood by which nearly all businesses operate. Every company department has distinct roles to play in the success of a business, and they are usually only able to perform their functions correctly if they can obtain accurate, timely data from other parts of the business. The data itself can take a myriad of forms – customer orders, stock inventory, product prices, buying trends, sales figures, general ledger entries, customer churn data, external streaming information (e.g. share prices), employee data, image assets, marketing copy or indeed any other kind of business information. Whatever the type of data, there's a good chance that someone, somewhere in the business would be interested in getting their hands on it.

In practice, though, sharing data within a business never seems to be as straightforward as it ought to be. There are a variety of reasons for this, which we will now discuss.

Lack of embracement of standards for data manipulation in software applications

Since relational databases were first developed in the 1970s, they have been the business tool of choice for long term storage of data. Although occasionally threatened by newer technology (like object-oriented databases, or XML data stores) the vast majority of the world's data is still held in them.

Although international standards do exist for manipulating data in relational databases (e.g. ANSI SQL, ODBC), third-party database vendors have traditionally been slow to wholeheartedly embrace these standards, frequently preferring to promote proprietary implementations of the same or similar functionality. Most relational databases allow the use of some form of the declarative computer language SQL to manipulate the data they hold, but whilst the different implementations of SQL are broadly similar, database vendors have typically developed their own dialect of the language, particularly for the more sophisticated database operations, which means that SQL code written for one type of database would likely need to be adapted before it would run correctly on a different vendor's database product. Similarly, innovative database features have always started life as proprietary extensions to the standards, only getting absorbed into the standards as and when sufficient numbers of vendors also implemented them. After all, if data access became truly universally standardised, databases would then become something of a commodity, with businesses being instantly able to swap out their database if something better/cheaper came along - something database vendors are keen to try and avoid.

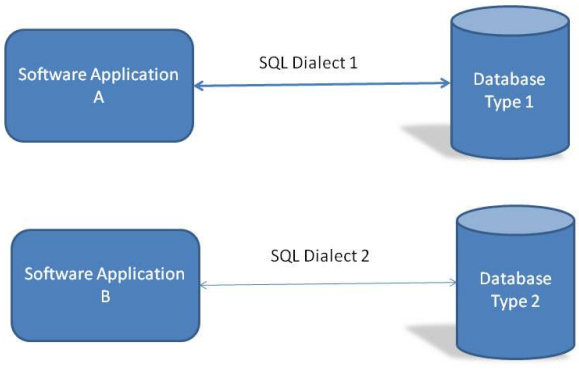

Figure 1 – Software code is dependent on the type of database used to hold its data

A common way in which customer 'lock in' is encouraged by database vendors is by the promotion of performing more data processing inside the database, rather than outside in software application code. For example, many database vendors offer 'stored procedures' as part of their product offering, which allows the development of potentially quite sophisticated business-specific APIs that give external access to the underlying database data. These stored procedures are code modules that are stored inside the database, and can include SQL commands for manipulating data, but unlike pure SQL, they aren't restricted to just fetching or inserting or deleting sets of data. Stored Procedures are typically written in a procedural programming language, and can process and evaluate that data, enforce business logic, restrict access to data for different users, and similar operations. By liberally employing stored procedures, a business can attempt to build a comprehensive data access layer that can give many different external applications a common way of accessing data from that database. The result is that, in many cases, business logic that might have been included as part of the external software applications needing the data has now been embedded in the database itself.

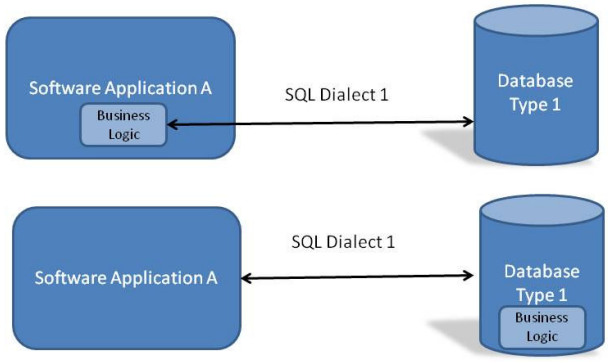

Figure 2 - Where to position business logic?

The lure of using stored procedures is that database vendors often claim that as well as giving a common way for external applications to access data, they give a performance advantage over pure SQL-type data manipulation. This is because such procedures are typically pre-compiled inside the database, and can have execution plans associated with them to help the database process the data in the fastest possible way. Ad hoc SQL commands from external software applications have to be interpreted and executed on the fly by the database. The down side, though, is that the procedural programming languages that are typically used to write these stored procedures are nearly always proprietary to the database vendor, meaning that if a business employs them heavily, and writes lots of API code in the database vendor's proprietary language, they are effectively 'locked in' to that particular vendor, and would find it very difficult to swap to use a different type of database without a complete rewrite of all the business logic associated with data access in that database.

So, what does all of this mean in practice? Well, it means that a lot of the available business software that manipulates data is quite tightly coupled to the type of database that stores that data. Such software has often been written from the ground up assuming that it will have permanent access to a particular type of relational database, and that it can use the specific features that this type of database has to offer. In others words, transferring such a business application to store data in a different type of database might involve a lot of work, and may in fact be impossible unless you control the source code for the application in question, or you can persuade the software vendor to undertake this work for you.

It's certainly still possible to write business software applications in a 'database neutral' way, but in practice most software vendors/in-house development teams don't bother – the extra work involved in writing software applications this way often can't be justified in the short

term. Commercial business applications are therefore often sold on the basis that they must be deployed on top of a certain third party vendor's database product, whilst in-house development teams writing data applications for the business often choose to target the type of database that the team is most familiar with, or perhaps the one that the company currently has spare licenses for.

So, in summary, the lack of embracement of standards for data access in the software industry means that many enterprise software applications that work with data are in fact quite tightly tied to the type of relational database they were originally designed to work with. For this reason, it's not typically possible to hold all of the company's data in 'one big database' that all business software applications can address.

Most corporate environments use an array of business software applications as part of their IT infrastructure– some perhaps purchased from third party vendors, some inherited legacy applications from older parts of the business, and some custom designed in-house to meet current business needs. An obvious result of the tying of individual business applications to their associated database is there may be many different databases in use within a company that store essentially the same sort of business data. For example, it might be that company's customers are split across several different databases depending on which services they take, or depending on when the customer signed up. If the company has previously been involved in corporate merger activities, there may be several sets of customers handled by completely different business systems.

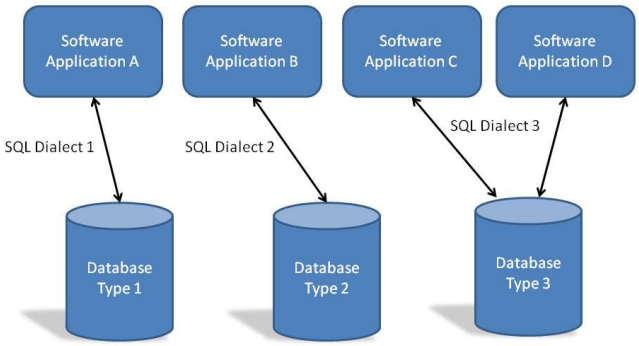

Figure 3 - Typical corporate IT system - lots of databases!

This is one of the largest impediments to effective flow of business data – having lots of application specific databases dotted around the business, perhaps owned by specific business departments, or servicing specific business applications, that are unable to effectively share their data with other business applications, and the rest of the company. Answering a simple question like 'How many customers do we have' can be difficult if the customers are spread over a dozen different databases.

Sometimes the isolation of data may be deliberate – there may be security or sensitivity reasons why data in some databases shouldn't be shared with other personnel and departments. But more often than not, it's more by accident than design – the business systems have simply developed organically that way, and when the business has grown to a certain size, it becomes a daunting task to try and alter the situation.

So, how have businesses attempted to solve this problem? Over time, many different approaches have been tried. A common one is the writing and deployment of reports. If there's data that you want to get hold of in another department's database, you can persuade them to write some code that will fetch the data that you want out of their database at periodic intervals, and then send it to you. You can then input the data into your business

systems (and underlying database) if you wish, and use the information as part of your business operations.

This approach can be effective, as long as there's no urgency for you to receive the very latest information. For example, the finance department might require quarterly sales totals from the sales department's database to process into the company's general ledger, but it perhaps isn't that important that the finance department receive the sales data the minute a deal is struck. Problems can arise, though, if there are frequent alterations to the report specifications in order to satisfy the recipient's new business goals. This can result in repeated requests for changes to existing reports, and new reports being asked for, which may not be prioritised by the department responsible for delivering them.

In many other business areas, however, it is becoming increasingly important to receive immediate notification of new data arriving in 'foreign' databases. Modern enterprises are often very complicated supply chains, where correct provisioning of services will only occur if the right data is received by the right department at the right time. This reduces the window significantly for operations to be successfully based around report generation, and makes real-time data update into foreign databases more of a necessity, rather than a luxury, for many businesses.

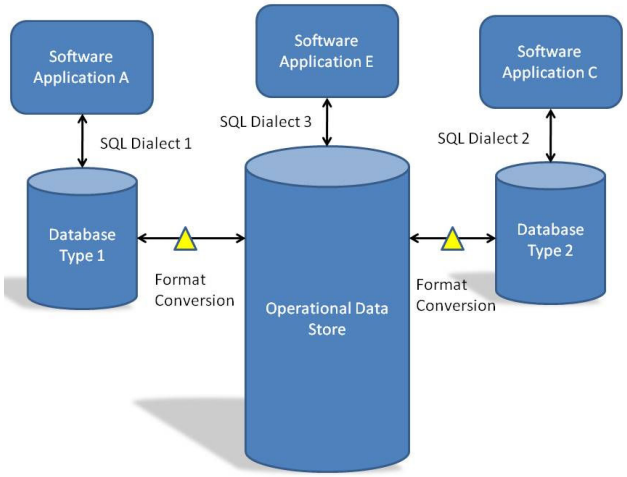

Figure 4 - Deployment of an Operational Data Store

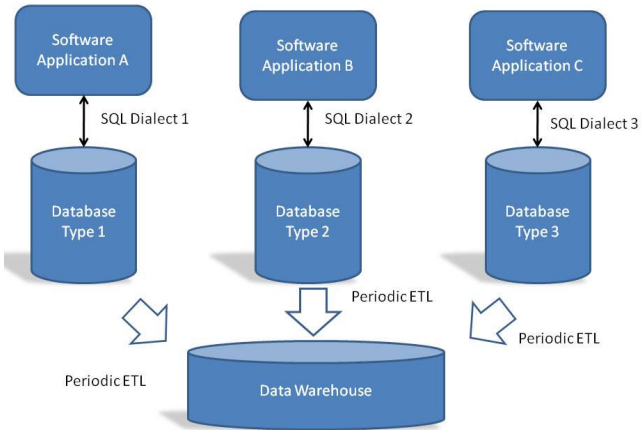

Figure 5 -Deployment of a Data Warehouse

Another, more sophisticated way of trying to address issues of data sharing is for the business to build an Operational Data Store or Data Warehouse, to which business applications export their relevant data at regular intervals, or perhaps in real-time. An Operational Data Store is intended to allow data from different business systems to be integrated and cleansed to give an overall picture of business operations at any given time, which may in turn be used by other business systems in making operational decisions, whereas a Data Warehouse usually follows ETL principles, periodically loading large volumes of historic data into another database to use in the analysis of trends. Data Warehouses are not typically involved with day-to-day running of the business.

Whilst Data Warehouses and Operational Data Stores can certainly help in the sharing of data within the company, the detail of how to build one, and how to merge data from different sources with different data formats can often prove somewhat problematic.

Lack of an agreed logical format for storing business data in databases

If non-adherence to data access standards has caused problems in terms of proliferation of lots of different small to medium-sized databases within the enterprise, the non-existence of standards for the logical formatting of data has arguably caused even more problems.

Once you've successfully located the data you're looking for, how do you transmit it to other interested parties within the company? The *physical* transmission part is easy enough – we've known how to network computers for a long time. It's the data *formatting* that's more problematic - in other words how will the user or the piece of software at the other end of the wire understand what it is we've sent them, and know how to interpret that data as part of their own activities?

This sort of activity is typically known as 'enterprise integration', and usually involves a large amount of mapping from one type of data to another. In just the same way that database vendors have been slow to agree on standards to physically access and manipulate data, application developers have been even slower to agree on any kind of standards about what constitutes the best data format for the business data that they individually hold. So, even simple and common concepts like addresses may be held in a completely different way in two different business applications, or more specifically in the databases that those applications store their data in. In order to pass data between two business systems, some software that understands both formats needs to be deployed, for example to translate one address format into the other.

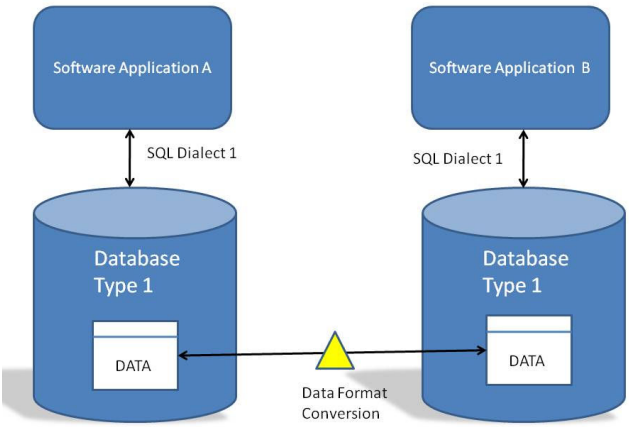

Figure 6 -Data Format Conversion

Why do several different business applications within a company need to hold essentially the same data? Well, this issue often arises because of an uncertainty by the software vendors over how their software application will eventually be deployed 'in the field'. Some business applications are sold to small businesses, who will go on to use them as their prime tool, and such businesses frequently own very few other software applications. They will expect that the application they've purchased is capable of storing all of the data it requires by itself, in its own local database. However, other (often larger) customers will require the same software application to interface with the other business systems they own, and to take some of its business data from sources elsewhere in the business. There are also sometimes issues of performance connected with large volumes of data – holding a certain amount of data locally may sometimes be the only way to achieve the performance characteristics the business expect of the application in question.

The usual compromise that is reached to meet all these deployment options is that enterprise applications store their working data in their own (often vendor specific) database, but that they use 'open APIs' to allow some of the data they need to originate from elsewhere, or to export their own data to external systems. In other words, there is always a local copy of the data, in a local database for the enterprise application to work with, but that data may in fact be a copy, and the copy may have come from data originating in some other business system within the company, or indeed from outside the company. Hence the reason why the same data can often be found in several different databases within the company, stored in different data formats.

The term 'open' in 'open APIs' refers to the fact that the APIs are publicly acknowledged and defined by the software vendor, and are made available to be used by other business applications. They are distinguished from 'closed APIs', which are used internally by the developers of the software, and are often connected to the internal workings of the software application. The fact that the API is 'open' means it is available for use by other external business applications, and that it delivers and accepts data in a format pre-defined by the software vendor. Unless a company's business software all comes from the same source, chances are that this data format isn't the same one that the other applications understand and accept, though, hence the need for additional 'plumbing code' to connect up business systems wanting to share data via 'open APIs' .

Unfortunately, this type of 'plumbing code' is notoriously fragile, with any kind of change in data format in the software applications sharing data requiring a re-engineering cycle. A data format change might arise, for example, from upgrading a particular software application to a newer version, or even simply by using the existing software in a new way.

Whilst this sort of formatting upgrade project is manageable when a small number of business applications share data, it often starts to become a real problem for larger enterprises, where even minor data format changes cascade through scores of business systems, requiring a full round of code changes, and then unit and integration testing from potentially multiple development teams. Often, this process starts to become a real impediment to change within the business, and creates a limit to how responsive the business can be to the marketplace. After all, most data format changes are requested in order to meet some sort of business goal.

The result is that data format changes are often postponed as long as possible, or a collection of them are rolled into a larger re-engineering project, resulting in a business-wide 'all or nothing' approach to enhancements of the company's business systems. It is well known, however, that the larger the project, the more the risk of project failure, so resisting all data format changes for as long as possible can unfortunately lead to unmanageable projects that never deliver the functionality that the business is looking for.

What can be done?

Many businesses operating multiple databases, each storing some portion of the company's data, would ideally like to be able to obtain a 'single view' of all the company's data. They can envisage a state where it doesn't matter where any particular piece of data is physically stored, that this data is still accessible to any other business system that needs it or is authorised to see it. They can imagine a situation where the relationships between that data, and other data, perhaps held in entirely different business systems, are immediately apparent. They would like an infrastructure where data is automatically passed around to the systems that need it and is updated in real- time. They would like business systems where data definitions can be changed without requiring extensive re-engineering of all applications that come into contact with that data, and without requiring extensive system down time. This includes modification to the underlying databases.

To achieve something like this typically means abstracting the data model to a higher level than the individual database schema. This is because no particular database in the company has the complete picture of the company's data. This also highlights another flaw with embedding business logic inside the database via stored procedures – if the business data is spread over multiple databases, that likely means the business logic associated with it must be too. This leads to the business logic being replicated inside different databases, and potentially becoming differentiated over time, if business logic is later updated in some places, and not in others. If the business logic is abstracted to a higher level along with the data model, that allows the same business logic to potentially be deployed across multiple applications, and for it to be managed centrally. We will call this higher level of abstraction the 'object' level.

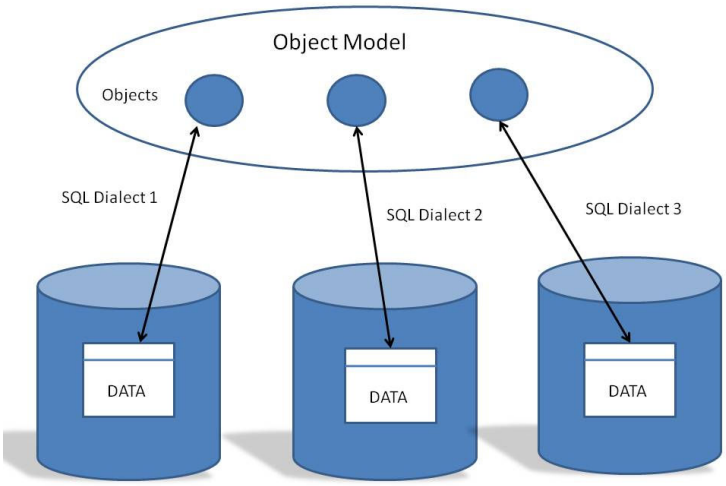

Figure 7 - Abstraction of an object model

The term 'object' means many different things to different people, but the most common interpretation of the word 'object' in software comes from the software paradigm known as 'Object-Oriented (OO) programming' which rose to dominance in the 1990s. Most of the software applications that process data these days are written in an Object-Oriented programming language (e.g. C#, Java, C++), so it makes sense to use the term 'object' in the sense understood by the programming languages that will most likely be manipulating the data being retrieved from the company's databases.

If we want to abstract our data model to the 'object' level, so that our database data can easily be processed in software applications written using modern object-oriented programming languages, one obstacle we need to overcome is the so-called 'Impedance

Mismatch' between an object model, as the concept is understood from OO languages, and the data structures used to hold data in relational databases, which were designed in the 70s.

An easy way to see one manifestation of the problem is to consider the concept of inheritance, which is familiar from OO programming. In OO programming languages, objects are regarded as instances of classes, where classes can form parent-child class hierarchies. Child objects are considered to be valid instances of their parent class, as well as valid instances of their own class. Taking a simplified illustrative example from biology, 'cat' and 'dog' are both valid examples of the parent class of 'mammal', which in turn is a valid example of parent class 'animal', which in turn is a valid example of parent class 'organism'. All properties of the classes 'organism', 'animal' and 'mammal' are present in both 'cat' and 'dog'.

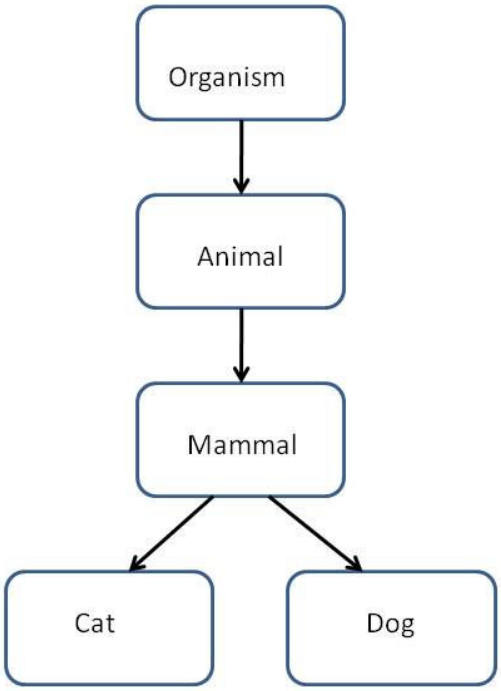

Figure 8 - Simple Class Hierarchy

We might choose to store information about particular 'cats', 'dogs', 'animals', 'mammals' and 'organisms' in a set of tables in a relational database, but there is no simple way to model the above OO class hierarchy in a relational database in such a way that it is immediately obvious that cats and dogs are both valid examples of animals and of mammals and of organisms, and that a data request to find all mammals should therefore return all cats and all dogs.

The industry term for a solution that attempts to bridge the 'Impedance Mismatch' is an Object-Relational Mapping Tool (or ORM), and any framework that attempts to abstract things to the 'object' level should include this functionality, to allow the object-oriented programming languages that are typically used to build business software applications to interface in a natural fashion with relational databases.

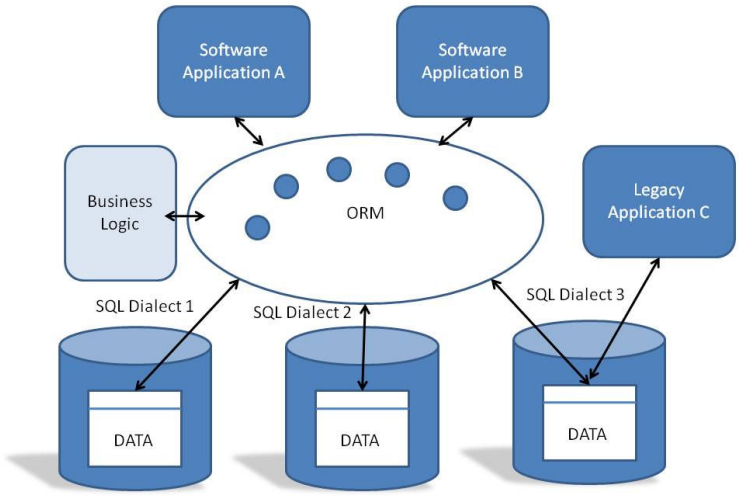

Figure 9 -Deployment of an ORM

As well as mapping between OO and relational concepts, if the company's data is physically stored in several different vendors' databases, a

framework that can communicate and speak the dialect of databases from different vendors is also essential.

So, a step towards a solution is to build an object model for the whole of the company's data, which expresses the relationships between the different types of data that the company holds across the business. Whether this object model is physically realized is a design decision - if it is, then this will result in something like the Operational Data Store mentioned earlier, whereas if it isn't, the object model can be mapped directly onto the data in the distributed databases, or alternatively onto a data access layer or set of stored procedures, if direct data access is not appropriate/available.

But how can the fragility of the 'plumbing code' be avoided, making the interfaces between systems more resilient, and more amenable to change? That depends on the extent to which the business controls the interface, and the two business systems on either side. In the best case scenario, where the company has access to the underlying source code of both applications, employment of modern programming techniques such as code generation and reflection can allow the creation of essentially auto-generated APIs that adapt to changes in the underlying object model, making the impact of data changes across the connected systems much less.

So, what does a company ultimately gain by investing in this sort of data infrastructure? If we take the example mentioned earlier of customers spread across a variety of legacy business systems, we could use this approach to build an object hierarchy that classifies each of type of legacy customer, and then uses the ORM to map those customer objects back to the business system that holds information about that type of customer. After creating such an object model, we can build new business applications that address *all* the company's customers and treat them in a common fashion, not just those held by a particular business system. This might allow the company to build (for example) a web portal that enables self-care for *all* the company's customers, or a software application to help customer service operatives to help all the company's customers in a call centre.

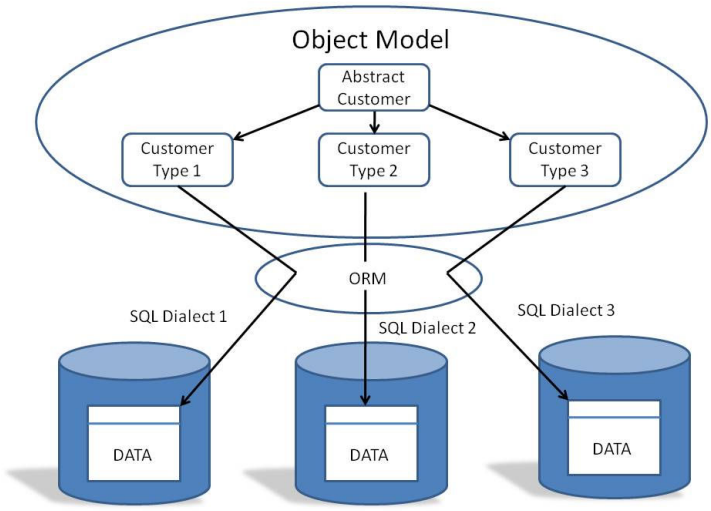

Figure 10 - Creation of a 'single view' for customers

Opening up the whole of the company's data in a 'single view' opens up all sort of possibilities, for example cross-selling of new products to old customers, better responsivity of the business to customer enquiries, better analysis of customer trends, improvements in overall data quality and easier revenue assurance. It also leaves the business less at the mercy of the IT architecture it has inherited from its past.

Conclusion

The ease by which data flows around an enterprise is related to the number of databases being used by business software applications, the type of those databases, and the format of data being held in those databases. Avoiding placing business logic inside databases can help make data more portable. Enhanced information flow around the business can be achieved by abstracting the data model to a level higher than the database schema ('object level'), and mapping the resulting object model back to data held in individual databases or data layers, or to an Operational Data Store. This can be used to produce a 'single view' of company data. These data maps can be made more resilient to change if the business controls both sides of the interface, by employing advanced programming techniques. This can lead to advantages for the business in terms of responsiveness, immediate availability of a complete set of information and improved adaptivity to change.

Further Information

For further information on the DATRICA data management framework, and about how it can help your business overcome data flow problems, please contact enquiries@silhouettesolutions.com

Glossary

ANSI	The **A**merican **N**ational **S**tandards **I**nstitute. A non-profit organisation promoting standards in industry.
API	**A**pplication **P**rogramming **I**nterface. A defined method of interacting with a particular service, often used to obtain certain types of data. Normally abstracted from the physical storage of the data itself.
ETL	**E**xtract, **T**ransform and **L**oad. Technique typically used to load data from business systems into Data Warehouses.
ODBC	**O**pen **D**ata**b**ase **C**onnectivity. A standard API for addressing relational databases.
ODS	**O**perational **D**ata **S**tore. A data repository usually assembled from data originating from other business systems, and used to give a wider view of this data, where direct access to the data would be inappropriate.
OO	**O**bject **O**riented. A programming paradigm originally invented in the 1960s, which achieved dominance in the 1990s.
ORM	**O**bject **R**elational **M**apping tool. A method of translating between the paradigms used in the object- oriented programming model, and the paradigms used in the relational database model.
Relational Database	A database built according to the relational model, pioneered by E.F.Codd in the 1970s. Other types of databases exist. Most of the popular commercial database products in use in business at the current time are primarily relational databases, although many now incorporate some object-oriented or hybrid features.
SQL	**S**tructured **Q**uery **L**anguage. Declarative programming language used for manipulating data in relational databases. Many dialects of SQL exist, tied to different database products.
XML	E**x**tensible **M**arkup **L**anguage. A subset of SGML, designed to allow easy transmission of user defined data structures between different business systems

References

Building the Operational Data Store (Second Edition) W.H. Inmon John Wiley & Sons