

## **Why Objects? And Why an Object Database?**

**Jasmine Conference, CA-World99**

**session: JG106S**

**V1.10, March 1999.**

*Just what are the benefits of object orientation for application development? What does an object database give me that my relational database system doesn't now? Should I change or add to my database environment? I mean, surely it's not relevant to me?*

*This session examines object-oriented techniques, and their place in development today. It focuses on the part of the development cycle where we commonly split into separate program specification and database specification teams, and how this will differ with an object database such as Jasmine. What are the benefits?*

### **In the Beginning...**

The computing industry started out years ago as a collection of technowienies who performed their black magic out of sheer effort and wizardry. Attempts to make programming and system development more of a science and less of an art came up with **Structured Programming**, and its sister techniques in analysis and design. This recognised that most programs consist of similar elements: **Code** and **Data**. Thus structured analysis and design sought to identify and separate out the code and data elements from a problem so they could be turned into programs (and databases).

### **Structured Programming?**

So lets look at structured (programming) techniques. These insist that software construction is done from the top-down, meaning that we design software by first writing a topmost function. From there, one after another, functions call other functions and branch out until they are repeatedly broken down to the lowest level required to actually do our task in the programming language being used. From the simple DO command, a topmost task starts a chain reaction of code which offers menus and data screens. Eventually, the code trickles down to a specific act, task, or feat that is undertaken by a single procedure or function.

This technique has its roots in **reductionism**; the idea that any problem is solvable and definable if you break it down to its smallest elements.

This is the way of the past for one important reason: Windows and GUI, multi-tasking operating systems. Consider the DOS applications you ran in the past. You entered an EXE filename at the DOS prompt, and the program started. It continued running until you had performed whatever you needed to do, and selected Quit. At no time could the PC do any other task (note that I'm ignoring TSR programs for simplicity here!) until that one finished. Top-down structured programming was designed for this environment, a task starts, and works its way down the execution tree of the program until it stops.

But what of Windows? As I write this, I have Microsoft Word open, with two copies of this document in front of me, one containing my changes, and another with a colleagues comments. I have the Windows Explorer running behind ready to locate files for me, a CD player program running and Norton Utilities monitoring my PC's overall operation. I just switched to the CD program, adjusted the volume, and switched back to Word. Where is the top-down processing here? The answer is that there is none, at least none like we used to see in DOS programs. Modern operating systems are based on an **Events**, that is they are **Event-Oriented**.

I clicked on the icon for the CD player program. That mouse "click" is an **event**. Windows tells the CD program that an event occurred, and asks it what it wants to do about it. The program responded by displaying a menu. Windows also told Word that my attention had wandered, and not to bother displaying the flashing cursor where I was just editing text. The colour of Word's title bar also changes, echoing my wandering mood. Back to the CD program. It's displayed a menu. I select the Volume option, and use the mouse to adjust a slider down slightly to reduce the volume. I click on Close, and the click back on my Word document. Inside Windows, that simple operation involved a flurry of events and calls between Windows and the various programs. But the only program that was running the whole time was Windows itself. Word, the CD program, and the others aren't doing anything most of the time, they are simply waiting for Windows to call them up and tell them the user did something of relevance. Ideally, a small piece of the program is executed in response to each event, and then Windows resumes its watch.

There are two important issues here: A program is no longer a single entity that starts executing, and continues until its job is complete. It's now a composition of smaller fragments of programs that execute on request, and then stop. Secondly, these fragments no longer execute in a fixed sequence. Today I opened Word, and then Clicked on the documents at the bottom of the File menu that I was looking at last time. Next time I open Word I may select the New option to create a document, or I may do things in various combinations, with several documents open at once. The point is that Word does not know what I will do next, but it has to cope with my whims, no matter what they are!

The second issue is obvious – **complexity**. This almost drove C programmers nuts when the early versions of Windows were released. The advantages and flexibility provided to the user are obvious, but the programmer has a nearly impossible task trying to code this using just structured techniques. Sanity was restored when designers and programmers began to learn how object orientation gave them a new perspective on programming Windows applications. Object orientation demands we treat everything on the Windows screen, and the matching elements in our programs as objects. That is, as individual entities that can be described as having certain **characteristics** and **behaviour**.

For instance, the mouse cursor (the arrow) knows that it's an arrow shape. When told that the mouse has moved, it moves in the same direction. When told that it is over a text entry box, it changes shape to the vertical "I" bar. When a program is performing a long operation, it tells the cursor to become an hourglass. If the mouse cursor is an **object**, it has characteristics that define it (its shape and position on the screen). It responds to events with certain predefined behaviour (moving position, changing shape). Most importantly, it does all of this irrespective of what is happening around it.

Using this model, the programmer is able to make sense out of the complexity that confronts him or her. The whole environment that is a Windows program can be built by constructing the elements individually, making sure they are robust, and unaffected by the actions of other objects around them. But I am getting ahead of myself. I wanted to describe terminology before getting too deep in object orientated techniques.

Getting back to my comment on change earlier. If a program is constructed of objects, any change that is desired by the client will result in some property or behaviour of an object being changed. But if the rest of the objects in the application are constructed properly, then the change will become operational with a minimum of fuss. If we need to construct another application in the future that performs similar operations to this one, we can simply use or copy the objects we use to construct it.

Of course, what I have just described is only of interest to the programmers out there. The real power of object orientation is its ability to model business problems and processes, and then carry that model into the technical phases of a project with little alteration. It's a paradigm that comes naturally.

## **Relational Databases**

Before we get further into this object stuff, it's important that we examine that other bastion of programming today:

**Relational Databases**. Invented in the 1970's, relational technology grew slowly during the 1980's as software companies experimented with the concepts, and standards committees debated the benefits of this SQL syntax over that. It exploded during the early 1990's as most relational database products matured and **client-server** development became trendy.

Today most companies speak of database development and relational Databases (RDBMS) in the same breath. In fact, only computer science students will recall that there are two other database types: **Network** and **Hierarchical** (somehow, Object Databases never entered such university discussions?). How and why did relational databases become so ubiquitous?

The reasons are two:

1. They do what they do pretty well – **Maturity**.
2. The design techniques required to build one are almost a science, not an art – **Normalisation**.

As we will discuss later, relational databases are exceptionally good at number crunching. Depending on the product, custom software, and how fine tuned your installation is, they can also achieve very high transaction rates. Related issues such as transaction logging, administration tools, database performance tuning are all comprehensively dealt with. All of the major vendors are into their umpteenth version of their flagship database.; many with several complete rewrites to take advantage of the latest processor and hardware technologies.

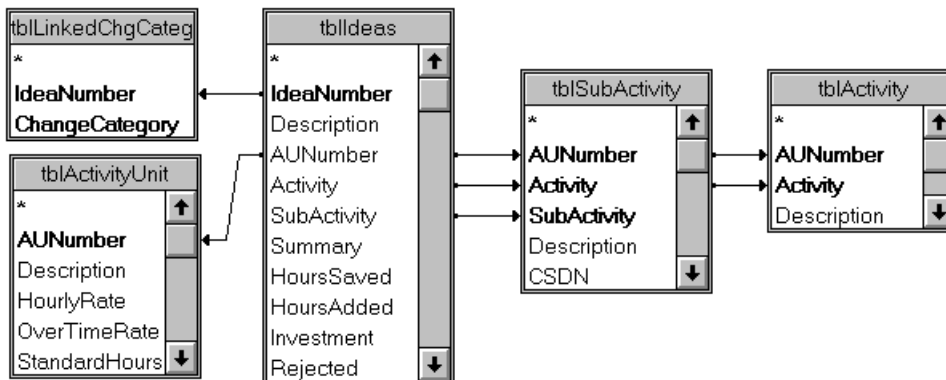
Database **Normalisation** is the process of taking your users basic data requirements (lists of names, details and information they need to store), and turning that into a complete relational database design. Assuming you got your business requirements right, the process normalisation virtually guarantees to produce a relational database design with every thing tagged, keyed, indexed and in the correct place. Data is arranged for zero redundancy, primary and foreign keys are immediately obvious. From this it's a simple step to create a relational database schema.

Relational databases get their name from the relations defined between tables in a database. Relations are defined where two or more tables can be combined in some logical fashion, where a row on one table can be matched to one or more rows in another. A relational database gets its name from the fact that the database engine can be made responsible for maintaining such relationships between tables. That is, if changes are made to one row, change must be made to all related rows in other tables; or changes are not allowed unless they are first done to the related rows. This represented the first popular database design which proposed to join raw data with some form of rules about how the data should be maintained.

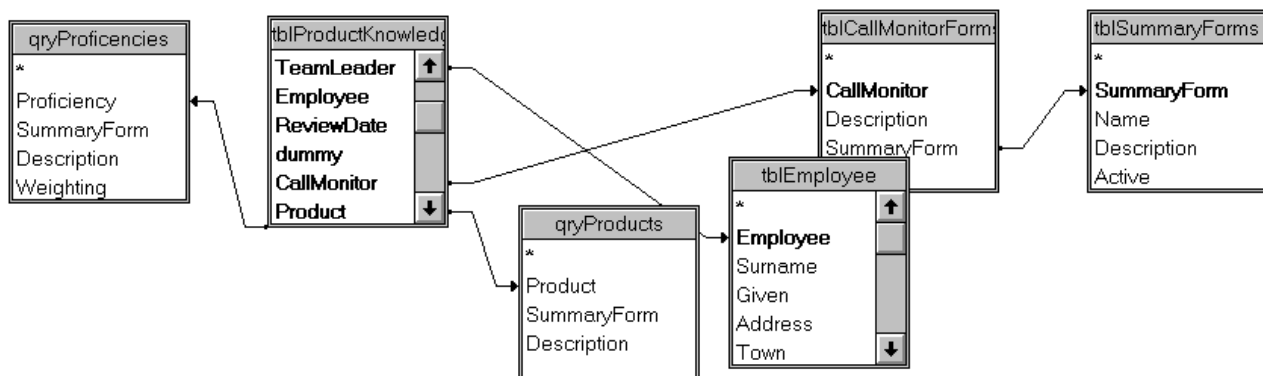
There are two basic relationships in a relational database. **One to many**, and **many to many**. When data is extracted from a database using **Structured Query Language** (SQL), one or more of these relationships is generally invoked through **joins** between related tables.

Up to this point, the story of relational databases has been a positive one. Joins, and SQL are where the complexity starts. In even a small database, you may need several joins in a single query to extract the data required. Figures 1 and 2 show typical examples of this. After a few joins are used, even visual query editors struggle to represent what is happening.

Another problem is that joins are slow. The more of them in a query, even with well applied indexes, the slower it will execute. This difference between constructing a query one way and another can be quite dramatic, and it's the reason why good SQL programmers and database administrators get paid what they do.



**Figure 1. Trying to regroup data that has been normalised.**



**Figure 2. As soon as you want to do something complex, the required joins become rather awkward.**

To combat the speed problems introduced by joins and managing relational integrity these databases were among the first to separate of client processing from server activity. That is, the client program was responsible for the user interface, and basic algorithms, while the server handled all data related activities. The main activity of course was query processing and bulk data handling. Note that this is different from two main application types at the time: host terminal systems (eg: CICS) where the central processor was responsible for everything, and networked PC system which shared data on a file server, but did all processing locally on client PCs.

As such, **client server** development, and **application partitioning** have been hot topics for many years now. And, you rarely see these terms without relational databases being mentioned!

It is very important to realise that we will not be putting our relational databases aside any time soon. The emphasis in this paper is to **pick the best tool for the job**, and to give you some pointers on how to decide between these technologies for any particular project.

## Object Oriented Analysis and Design

Structured analysis concentrates on separating data and functionality. The flow of data and the functions that operate on it are charted and described *separately*. Common structured tools and processes such as dataflow diagrams, entity diagrams, and top down decomposition will be familiar to many of you.

Object Oriented Analysis has a different focus however; it looks to the *combination of data and its associated behaviour*. I think the OO approach is a more holistic one – it tries to look at the business problem as a single entity. Any units we break the problem into should closely represent real world objects. Data and or behaviour related to the same conceptual unit should be kept together because that's how we naturally tend to think of them.

Before I can discuss the benefits of these techniques, lets review them briefly:

## Analysis

Object oriented techniques follow a similar order to structured techniques: analysis first, then design, then programming. The order is important, as each builds on the results of the prior phase.

Analysis is concerned with **What** has to be done, and nothing else. Insights into **How** we might go about achieving this are to be noted down so they aren't lost, but they are not the focus of this investigation. This process is firstly about (business) requirements analysis, with technical needs a distant second. The business client is closely involved at this time. As a result, a final report should be kept in business terminology where possible. Note that we should avoid business jargon as much as technical/computer jargon. A glossary of terms is frequently a useful section near the start of an analysis report.

Analysis consists of two steps:

1. **Gathering** as much information as is practical about the business problem (or **Problem Domain**). You often see this discussed under the wider heading of **Business Requirements Analysis**.
2. Attempting to **classify** the information. That is, identify the classes and objects inherent within the descriptions the first step provides and thence the relationships between them. We should avoid too much detail here, capturing the overall picture is the primary goal. Details will be worked out during the design phase.

The result should be a high level model of the business process we need to automate. This will include:

- A description of the basic business requirements.
- Descriptions of the Classes in the system. Remember that a **class** is the generalised blueprint used to create individual objects at runtime. All details must be kept in business terminology and remember that the discussion will be platform neutral.
- An idea of how all classes in the model relate to each other. That is, most classes will either have common elements with others, or they will be used by other classes to achieve their purpose. If classes have similarities, then they may have an **inheritance** relationship. If one class uses another to perform a task, then they have a **containing** relationship.
- Creating a **Class Hierarchy** by documenting the inheritance relationships.
- Creating an **Object Hierarchy** by documenting the containing relationships.

## Design

If analysis works out what we want to achieve, design looks at **How** we will implement our proposed solution. To this end we have a number of tasks to perform:

- *Classification of all objects* identified during Analysis. Find the Classes behind the objects, and try to document how they interact and relate to each other.
- *Filling in missing details* on the properties and methods of all Classes. Our focus is entirely on how our classes will perform their actions and behaviour, and how their properties will be manipulated.
- Consulting with *technical* staff, begin to add in the classes required for programming, operations, environment and database implementation. These classes will support the operation of the business model, but may not interact with it directly (eg: application, windows, menus and classes for other screen elements, or abstract concepts such as files).
- Start thinking about *database design* – whether you will be using an Object database, or a Relational one. Note that this comes at the end of the design process, and must be a separate step if you elect to use a relational database.

It's possible to create a wealth of documentation if you adhere to the above strictly. For all but the largest projects this may be unnecessary – but if you must keep paperwork to a minimum, make sure you have the following firmly documented and understood by all parties:

- The **key abstractions** in our model. The key items that relate many of the objects identified during analysis together. We have to pin these down and describe them. They become the linchpins, or the *base classes* of the Class and Object Hierarchies we will construct during design. Everyone must understand the business model you are implementing.
- The **principle mechanisms** at work in each class. That is, the main process or purpose behind each object, what it's supposed to do, and why it exists. Your technical staff will be able to deal with most minor problems as they encounter them during development, so identify only the main technology and design areas that are critical to the success of the project.

Even though design deals with technology, and the technical side of producing a solution, all documentation should be able to look the original business terminology for class descriptions, actions, and relationships. Technical implementation details will slide into this framework with no problem as long as we keep our designs in that single OO framework.

## Relationships

One key point that I must explain is **object oriented relationships**. As indicated above, analysis and design define both the objects in our business problem, and the relationships between them. If we were performing structured (data) analysis, we would end up with list of table definitions, and the relationships between each table expressed as one:many and many:many relations.

In object oriented designs there are two basic kinds of relationships between objects (or classes):

### Inheritance Relationships

This relationship is based on **inheritance**, the idea that you define one class as being the same as another, but with some differences. This is identical to a child inheriting the basic family characteristics from his or her parents. You can generally recognise the familial relation between parents and children, even though the children will differ in some ways. In the same way, a **child class** is the same as its **parent class**, with defined changes.

In general the *parent is the more general, or simpler class*, both in behaviour, and in properties. The child class always represents specialisation, the addition of behaviour and complexity.

There are two types of inheritance, *single* and *multiple*.

- **Single inheritance** is the most common. A child class inherits its characteristics from one parent class.
- **Multiple inheritance** is less common. This is due to few languages supporting it, and because it is rarely needed (though this is a subjective decision). Multiple inheritance is what it sounds like. A child class inherits behaviour and characteristics from two or more parent classes. Difficulties can arise if both parents have properties or methods using the same names – the compiler has to decide which take precedence.

Many times when multiple inheritance seems necessary, it can be satisfied entirely by using a **delegation** relationships, which is an **Is part of** relationship.

In a relational system, you may have a table which has a large number of fields which are mostly filled with one default value, but in special circumstances contain other values which have special meaning. Here most of the table's contents could be classified as belonging to the parent class, with the small number of exceptions being instances of one or more child classes.

### Containing Relationships

In this relationship, one object *contains* another (or a **reference** to another object). That is, the hosting object A, has a property that is of the type object B. The two work together to perform operations, either publicly, or internally (which is where the term *collaborative* comes from). A typical example is a Screen Form in Windows. A Form contains controls such as text boxes, list boxes, treeviews... etc. Each of these is an object in its own right, with defined behaviour and properties. The Window which hosts these objects/controls is known as a container class. The control objects are seen as being *part of* the window object.

There are two basic types of is part of relationships, and a third which is a little more difficult to pin down.

1. **Delegation**. This where one object contains another, but the host acts as though the contained object is part of itself. That is, it acts as though the methods and properties of the contained object are its own. In essence, it is *publicly* declaring the relationship, and opening it to outside control.
2. **Using**. Or a *Employer/employee* relationship. This is where one class requires the specific services of another, and usually hides the interface and relationship from outside operations. The host class in this case does not permit direct interaction with the contained classes, or only through tightly defined interfaces. When you add a control to a form in Windows, that's the last you have to worry about it – the form takes responsibility for it from that point on. If you need to use that control directly you wither cannot do it, or have to use a defined interface (depending on the language implementation). This is the most common of the *Is Part of* relationships you will see.
3. **Customer/Salesperson**. The name is perhaps misleading, but it refers to objects that are passed as *parameters* to or from methods of another object. These objects are not necessarily contained within the host object permanently (ie: they are not part of its structure), but are used by it as it performs its essential tasks.

There are other classification schemes that author's have devised over the last few years for *is part of* relationships. Without exception, they take the above three and either simply give them different names, or sub-divide them up into more specific categories. If you understand the above, then you have enough to work with from day to day.

One vital point is that these contained objects don't have to be singular. A property of an object may be a list of items such as a list of alternate addresses. This list could contain one item, no items, or many items. The list could be references to other objects as well. Thus a **Student** class could have a property **ClassesEnrolledIn**, which is a

list of references to instance of the **Course** class. We have just defined what would be one:many relationship in a relational database.

To take this a step further, our **Course** object can also have a property **StudentsEnrolled**, which is a list of Student objects enrolled in that class. We now have a many:many relationship!

With relationships, comes the need to maintain their integrity. It does no good to delete a **Student** object if we don't first remove that instance from the **StudentsEnrolled** lists in each **Course** object. This is one important focus of methods in a real object database system.

## Object Databases vs Relational Databases

The results of our Object Oriented Design are used directly in our application coding. Classes on paper become classes in our code – and discussions with the client on progress are easy, as we can point to the OOD documentation and indicate directly what has been accomplished to date (in terms that are meaningful to them).

The database design is another question however.

Before we start comparing and contrasting, please remember one very important point. **Object databases do not supersede Relational databases**. Each are designed to perform certain tasks very well (and the grey areas around the edges are mostly due to the experience of your programmers!). Use the rules-of-thumb below as a guide to which may be the better solution in any given scenario. At the same time, don't confuse complexity due to scale/size with the difficulties I've listed – a large system is going to be hard to implement no matter what the database!

### **Relational Databases are Better...?**

Say you are designing a classic order entry invoice system. I expect that the final object designs, and thence database design will fall into place without too much pain and anguish. In fact the class design will probably mirror the relational (table) design fairly closely. It doesn't get much simpler than this. If the results of relational data analysis and normalisation roughly equate to the object designs you got from object oriented analysis and design, then a relational database is probably a good move.

Any differences will be due to the relational model requiring a lot of unique primary key fields, foreign keys and generally fields to identify things where the object design simply used an object reference. Properties which were collections or lists of items become related tables, and so forth.

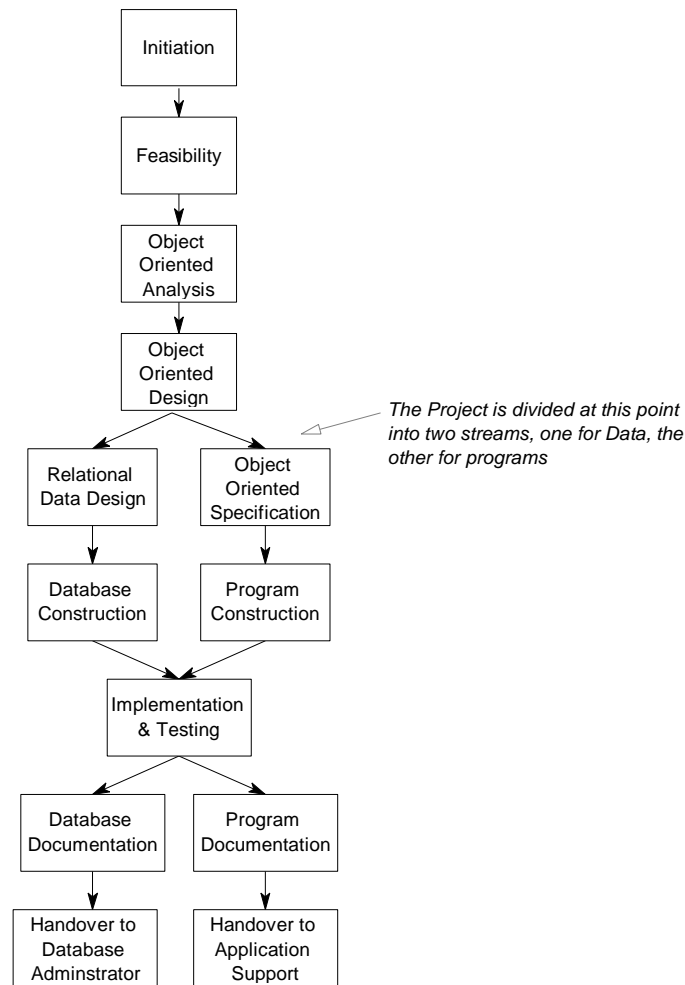
Your project plan is now going to look something like figure 3. After the main design phase, the project breaks into two streams, data design and program design. Obviously its vital to keep these two teams talking to each other, especially if the inevitable scope change occurs.

In short, relational technology is powerful in the following situations:

- When you apply relational data design, the table structures align closely with the class structures from your object designs. This means that with an object oriented application, the effort required to load data from tables into runtime objects for manipulation is minimal. Your application development team doesn't have to waste time writing a lot of "mapping" code.
- High number crunching load. Most of the data involved is numbers and manageable pieces of text, and you process it a lot during normal activity. A high transaction rate expected (though some OO database vendors may argue that last point).
- "No-one ever got fired for buying SQL". It's a sign of mature technology when even you senior managers recognise the names of tools. It may or may not be a good technical choice, but you aren't going to be questioned from above about the decision.

Now, before I get shot down in flames by the purists here, I'm not defending this line of reasoning, I just wanted to say it exists. Haven't we all seen examples of misapplied technology – and when you questioned the decision that led to that point, you find out that they wanted to go for the "known" product, as opposed to the "best" tool?!

- Win3.x, DOS or terminal client required. This is getting less and less of an issue, but it may be relevant to you. In the case of Jasmine, only Win95, NT 4 and Unix clients are supported.
- Win95 stand-alone solution required (no NT or Unix server available). Similar to the above, Jasmine is a server database. The only stand-alone solution possible requires NT 4.0 workstation.
- Reporting needs. Unfortunately, most of the better reporting solutions available today require a relational database. Some of the better tools only offer limited support for OLEDB, which is the only realistic way to report from an object database, save a direct interface. The only exceptions to this are a few CA-partners working on reporting tools.



**Figure 3: A typical OO project using a Relational Database**

### **Object Databases are Better... ?**

The first point above described when the relational design looked very similar to the object designs. Frequently that isn't the case, and we find ourselves struggling with the database design component of a project. Have you ever had a system where one or more of the following has occurred:

- There has been a high percentage of multimedia data types, or binary data (eg: whole files).
- The final data design contains many small tables, generally related in aspect, and all with structures that are highly similar, but not quite the same. Designing screens and reports for these is tedious because they all share a common basic structure, but there are enough differences to force different screens, code... etc for each.
- The data design has been extremely difficult to arrive at, and there are problems achieving consensus with widely different views on a suitable (normalised) database design among team members.
- The data design is forced to depend on a number of crucial programming "tricks" to work. There are few areas where a standard (individual) SQL query is sufficient to extract the data required without programming intervention.
- Numerous many:many relationships. These are always difficult to create and maintain as you need extra tables and procedures. Extra joins are always required in any query, which slows down data access times.
- A majority of regularly used queries involve either umpteen joins, cross-tabulation (pivots) or other structures both complex and slow in SQL.
- Sheer complexity. Relational databases are difficult to handle and document as they get more and more complex. If the original problem is extremely complex, simple data relationships don't solve the business problem, and program code becomes the only way many tables are linked, filled or used. OO techniques help with the program side, but the database remains a problem.

- Any combination of the above, which results in a project only being feasible or achievable if you get your top, top developers on it, who are familiar with bleeding-edge technology. Even if you proceed, there are expected problems with maintenance and ongoing enhancements requiring the same top-notch staff.

I suggest that these are classic instances where you were perhaps forcing a round *Object* peg into a square *Relational* hole. The business objectives were probably straightforward; complex perhaps, but they could be described in english with no communication problems. There may even have been an original, elegant manual or paper based system that was replaced, but the final (relational) computer system was, at the very least, awkward, and never achieved the elegance or simple functionality of the old method!?

You may find it a useful exercise to return to one of those old systems, revisit the original analysis documents, and see what type of Object Oriented Analysis/Object Oriented Design you can come up with. You may be surprised.

At these times we should look to an Object Database. The reason – an Object Database’s “database design” is essentially the results of our OOA/OOD process! Using an Object Database leaves you time to deal with the complexity of the business problem at hand, which is what you should be focussed on.

## Database Warehouses

Any discussion on this topic should come with a warning: *Incoming Buzzwords and Cliches!!*

One of the fundamental reasons for creating database warehouses is so the end-user can get at their own data more easily. Whether for simple reporting, or because they want to use OLAP tools, many companies are looking at their huge databases (relational and others), and trying to rework them into something more useable, while keeping the production transaction processing systems online.

What does this activity entail most frequently? De-Normalising a completely normalised data design. Massaging data into a state where an end-user can create a query and report without requiring 6 or 7 different joins in exactly the right sequence. In other words, bringing the database design back to something that better resembles how the business perceives their data in the first place. Which is something that an object database is rather good at to start with.

As such, another excellent use for object databases is providing a simpler class structure over the top of an existing production system. This isn’t a magic bullet, and you still cannot just turn tables into classes and expect the result to make sense. But if your object database provides the ability to pull data from an SQL back end, there are some seriously interesting possibilities!

## Using an Object Oriented Database

To construct an Object Oriented Database (OODB), the results of Object Oriented Design such as classes, instances and methods are directly utilised. Classes become database *Classes*; Class relationships, especially “is kind of” relations allow us to define *Class hierarchies* and an inheritance structure in our database. Properties on paper become Instance properties, Methods become ODQL and API methods in the database.

In general, there is little difference between our **Logical** object models, and the **Physical** database schema. The real decisions we are left with revolve around three issues:

1. Which objects/classes in our design have to be **persistent**. That is, which must be in the database, and which are only required at runtime within our program code? If it’s data you previously stored in an SQL or DBF table, then its going to become a persistent class on the (Jasmine) Server. Any object only required at runtime can be safely defined solely within our application.
2. Which methods will be coded into the database classes (using ODQL or API calls), and which will be coded into the matching classes in our program code? This is a basic question of **partitioning**: what parts of the system should reside on the server, and which parts should reside on the client!? Anything you previously coded as stored procedures, or database triggers are prime candidates for server methods. Any data-intensive activity should be done in a method so it executes on the server, and does not drag huge amounts of data across your LAN.
3. Should we be using ODQL or API-level language code for our methods? If you require API programming you need to look for staff with C/C++ skills.

If we have problems other than these, we need to re-evaluate our analysis and design process. Just as a programming problem may highlight poor or inadequate designs, problems integrating the design into the object database may indicate similar issues.

Remember that there is a distinct difference between a poor design, and one that is complex simply because the business model is complex. The business/clients should be with you every step of the way now, as you are now using a modelling process they can easily follow with little explanation (you should still be using *their* terminology to describe these classes and objects!). If you have to *iterate* back to an earlier phase to sort out an issue, then they must be involved just as they were the first time.

Handover to maintenance and support staff will now be much easier, as object-oriented designs constructed early in the project are now consistent with the entire system, programs and data alike. Reading, and understanding the original business requirements, and then going onto the technical designs should be a much smoother transition for staff unfamiliar with the project. If you have used OO techniques well, you should even see a reduction in maintenance costs over time, as making changes to code or data will be a simpler process.

The **paradigm shift** from relational design to object oriented design for databases will take a retraining on the part of your staff, and some time for them to see the advantages, and just how the newer schema works. I call it a *paradigm shift*, as it requires technical staff to think of data storage in different terms – it's a change in the way they approach a problem and look for a solution. If your team is already experienced in OO techniques for coding, then you have an advantage.

This is identical to the shift required of programmers several years back when most mainstream programming languages (such as Clipper) began offering OO extensions, and they had to change from the older *structured programming* techniques to object oriented programming. Similarly the jump from DOS to Windows programming. There were, and still are, cries of “*I don't understand!*”, as each of us waited until the penny dropped, and we exclaimed “*ohhh, but that's so easy, that can't be right!*”.

The following table is a collection of analogies which are NOT perfect, and any purist will have problems with them. However, they will assist you to wrap your mind around where some of the concepts fit, compared to how you may work now with relational technology:

Object Database Components	Relational Equivalent	Discussion
Class	Table	
Class Family	<i>None</i>	There is no relational analogy for a Class Family, as it is specifically a Jasmine concept (based on Class Hierarchy). One analogy is if you keep separate Test and Production copies of your database and configuration or INI files, or separate business data and program data into different databases. These could be separate Class Families in Jasmine.
Object/Instance	Record/Row	
Instance Property	Field/Column	
Class Property	<i>None</i>	Class Properties are generic, class level values that are not part of any specific instance, common to the entire class. There is no relational equivalent, excepting deliberate constructs of data and stored procedures in separate reference tables.
Instance Method	Stored Procedure	Both of these are pieces of code which can affect the data stored in the database. Here the similarity end however, as Methods can affect information outside the database, but only only for a particular object/instance. Stored procedures can affect any data, but are limited to acting on the database itself.
Class Method	Stored Procedure	Like Class Properties, Class methods operate at a generic level – ie: they work on the Class rather than specific instances of it. Stored procedures can have a similar scope though this will vary from product to product.
ODQL	SQL	ODQL is far more powerful an language, and methods compile to C DLLs, so its very fast as well.
API language in methods	Program code in client applications.	

## The Benefits of OO Techniques

To conclude this paper, I'll try to get back on track, and mention a few benefits of OO technology.

For OO techniques to be beneficial, they need to satisfy two criteria: They must fit into the project management and Project structures you are using at the moment (that is, it won't detract from what you are doing now). Secondly, they should actually improve matters!

## OO and Project Methodology

It seems that every would-be consultant, lecturer or high powered business speaker has a unique project management or software development model to push. Most of these are variations on one another, though I won't try to untangle them for you now. (NB: If you are interested in reading about Software development and Project Management, I suggest you look at Steve McConnell's **Rapid Development**, from Microsoft Press).

In all of these models, there are one or more analysis, design, coding and database development steps. Nothing could be simpler –just replace their suggested techniques for each step with object oriented techniques. It doesn't matter whether you are using an old waterfall project methodology, or the latest in iterative, spiral or evolutionary delivery models.

Please note, the one thing you should **never** do is take the results of structured analysis, or relational data analysis and try to make that your object database schema, or object program design. It just doesn't work, and is the single most common cause of technology frustration. Often the people denouncing object technology the loudest, are those who tried just that, and could understand why it failed. If you want to trial object oriented techniques, then give them a fair chance, and use them correctly. If you have an existing relational system you'd like to try moving to an object database, then the first step is to go back a redo the original analysis and design using the OO approach.

### **Improving Matters...**

I won't try to push a line on how much time these techniques will save you, or the productivity gains that result, as that is marketing hype that has proven false, at least in terms of actual time savings. In some ways, the phrase "**object oriented**" has been a catch cry that many vendors used to proclaim their software, their development tools or their operating system, whether or not it was actually true. Hopefully after this course you will be able to look at products with a discerning eye, and distinguish operational fact from marketing fiction.

Some more realistic and tangible benefits are:

- Better communication between the business and technical staff throughout a project. If you are using OO analysis and design (and coding and database) then the terminology should remain the same for all elements of a project from start to finish. There should never be a time when the technical staff will refer to parts of the system that the business cannot identify and understand.
- An application model that more closely resembles the business it is supposed to "automate". This makes handover to maintenance staff easier, as the problems the system deals with will be intimately related to its very structure. Analysis and design documentation will remain largely valid, even if low level (technical) implementation has changed as the project developed.
- Object-Oriented techniques literally force you to properly conduct the analysis and design steps of a project. Once done, the gains throughout the project are large, as you have a far more exact idea of what it is you are trying to achieve. Note that this is not so much a productivity gain as it is a risk management technique, preventing reworking and errors due to poorly understood requirements.
- Generally speaking, object-oriented techniques allow you to grasp and deal with more complex requirements than traditional structured methods. See my discussion on the problem with relational design and complexity earlier.
- The methodology is largely language and product neutral, though final technical designs can only be done once you know which language and database system will be used.
- Yes, code reuse is real. But it applies when you have many (potentially) common business functions over many systems. Thus the benefits of reuse only come when you build the second, third... etc applications. This is the same whether you are talking about program code, Microsoft's COM/ActiveX architecture, CORBA or an Object Database.

### **Conclusion**

*I have six honest serving me  
(they taught me all I knew);  
Their names are What and Why and When  
and How and Where and Who.*

Rudyard Kipling, 1902

To summarise, as you consider your next project or review an older one, keep Kipling's comment in mind. Its paramount to understand what your needs are before you can properly evaluate a solution.

Object Oriented Analysis and Design techniques are essential in today's environment. Determining **what** you want to do, and researching **how** you will achieve gets any project off to a good start.

Object designs aren't concerned about **where** you build and implement a system. If your performance requirements need a large-scale server database, so be it. If you are building a program for small business, it will be a stand-alone affair. Methods in your design implemented as methods in a OO database, or stored procedures in a relational system, or client application code are still the same method.

Technology is changing so quickly now that **when** you build your system will have a large impact on your choices. As hardware gets faster, and OO database products more mature the object database solution becomes more and more attractive as a solution. The work CA is putting into Jasmine TND (Jasmine's next major release) is a case in point here.

**Who** should be involved here? As many people as possible! Staff trained in OO techniques can be used in many areas, on most projects irrespective of their size or scope. OO techniques fold into modern project management practices very

---

neatly. Note that project management, and all the baggage that goes with it is still the single most important factor in a project's success. OO techniques will greatly assist any project's technical success, but only good project management (and people) can ensure the system is built on time or on budget. Anyone who tells you otherwise is trying to sell you left handed screw-drivers (and as a left handed bloke, I think I can make a definitive statement about such things!)

The final point: **Why**. One of the single most difficult things to pass on from developer to maintenance programmer (to programmer to programmer... ) is why a particular task or problem was solved the way it was. This may be a question of database, platform, transaction design, or simply "why does the invoice report only print 6 items, can it print more?". Not knowing the answer to these questions harms all later maintenance and enhancement work (can you say **Total Cost of Ownership**?). Programmers innocently make mistakes that are technically correct, but which break fundamental business rules long forgotten (until the client notices their data is screwed!).

Development teams are expert in either *forgetting* to provide documentation, or just using an automated source documentation tool – and handing you the printout. If you try to locate any of the original requirements documents, they will bear no resemblance to the final system if structured analysis and design techniques were used (or worse, if the team jumped directly into database normalisation).

OO techniques are better at keeping this information about **why** much longer, as they focus on joining data and behaviour at the hip, and never separating them. There is another secret reason too – OO techniques can only be successful if you perform analysis and design properly, and if you do that, your project is immediately ahead by miles, and you have the core of all the technical documentation you'll need already written.

## Further reading

There are a large number of books available that cover OO techniques. If you are interested in reading about this, look for one that discusses Analysis and Design (rather than programming). Many are now offering sets of classes and designs already pre-worked as examples of OO techniques for common business tasks such as Accounts Payable/receivable, inventory, sales... etc. There are also a small number of books on Object Databases available. Some are product specific, others are more general. Lately it seems to be a favourite topic for university theses, judging from the requests on the comp.databases.object newsgroup.

Key OO analysis and design authors to look for are: Booch, Coad and Rumbough. Be sure to look through a book before you purchase it. Each has a different focus, some on theory, others on practical application, and each has their favourite methods and tools. If you are searching on the internet for materials, include the following keywords: diagramming, UML, and CASE Software.

If you want information on object oriented programming, then certainly you must focus on books specific to your development language and environment. Be aware that such books frequently offer a skewed view on OO techniques (Visual Basic books are a case in point here), which is why I mentioned the authors above first.

There are a number of tools which can assist the design process, generally by providing a diagramming scheme to represent the class and object hierarchies, and their activities and interactions. Some of these are newer OO tools, others are more CASE oriented. Check out the tools from Rational Software (<http://www.rational.com>), Aonix (<http://www.aonic.com>), Platinum Technologies (<http://www.platinum.com>), Popkin Software (<http://www.popkin.com>), Riverton Software (<http://www.riverton.com>), Siemens Nixdorf (<http://community.sni.com>), Supernova (<http://www.supernova.com>), Visio (<http://www.visio.com>), and Object International (<http://www.oi.com>), and others (Please note: I just scanned the pile of journals in my office to construct that list, so it's by no means complete!).

Object International does have a freeware version of their diagramming tool: **Playground** available. Other vendors may have similar products. Note that each will focus on their specific techniques.

## Author:

Peter Fallon is the director of Castle Software Australia Pty Ltd, a company providing custom software development and consulting. He has been developing and supporting PC database applications since 1986, ranging from small business systems to large corporate installations. **[www.castlesoftware.com.au](http://www.castlesoftware.com.au)**

Peter has spoken on topics covering both programming and project management at CA-Technicons 1992, 1993, and 1994 in the USA, and OO and Jasmine topics at IngresWorld 97, and Harmony 98 in Melbourne Australia. He was the editor of "Clippings", the magazine for Clipper User Groups in Australia and New Zealand for 3 years and Vice-President of the Clipper User Group (Melbourne) in 1992. Currently Peter is working on a number of Jasmine products and training services as well as various Visual Basic and Access development projects. He can be reached at **[peterfallon@castlesoftware.com.au](mailto:peterfallon@castlesoftware.com.au)**

## Copyright Notice

No part of this paper may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the authors. For information please contact: Castle Software Australia Pty Ltd.

## Trademark Acknowledgements

Jasmine™ is a trademark of Computer Associates, CA-Visual Objects 2.0® is registered trademark of Computer Associates, Microsoft® Visual Basic® is a registered trademark of Microsoft and Microsoft® Office 97™ is a trademark of Microsoft.

All other product names and services identified throughout these notes are trademarks or registered trademarks of their respective companies. They are used throughout these notes in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with the notes.