“Entity Beans” Table Interfaces or Software Components?

Phil Robinson

E-mail: lonsdale@iinet.net.au

Abstract

This paper briefly describes the J2EE platform. It then goes on to explore the potential “clash of cultures” between object oriented and information systems developers embodied in the J2EE platform. It is noted that that conflict can result when both groups believe that they have the “only valid view”. In fact, the conflict is unnecessary because both groups have the correct view. The problem is that neither the object oriented or information systems perspective provides the full picture.

The danger inherent in such a conflict is that information systems developers will try to use database designs as the basis for software component designs. Conversely, object oriented developers will try to use software component designs as the basis for database designs. The paper argues that conceptual models are required to aid communication between the two groups of developers and to provide a shared understanding of the problem domain. Conceptual models provide a good basis for both database and software component designs. The ideas are illustrated using a simple example.

A Brief Tour of J2EE Technology

When Java was first introduced to the world, it was promoted as the means to dress up web pages with dynamic content. Over time, interest in using Java to develop browser-based applets waned but Java has not faded away. Instead, it has been “re-invented” as a server-side programming language. The release of the Java 2 Enterprise Edition (J2EE) platform firmly established Java as a force in the information systems arena.

J2EE is built on top of the standard Java 2 platform. It consists of a set of specifications, application programming interfaces (APIs), and protocols intended to assist with the development of n-tiered, web-based

applications. The centre piece of J2EE is the Enterprise Java Bean (EJB) component model. EJBs are server-side components that are executed inside a "container" [1].

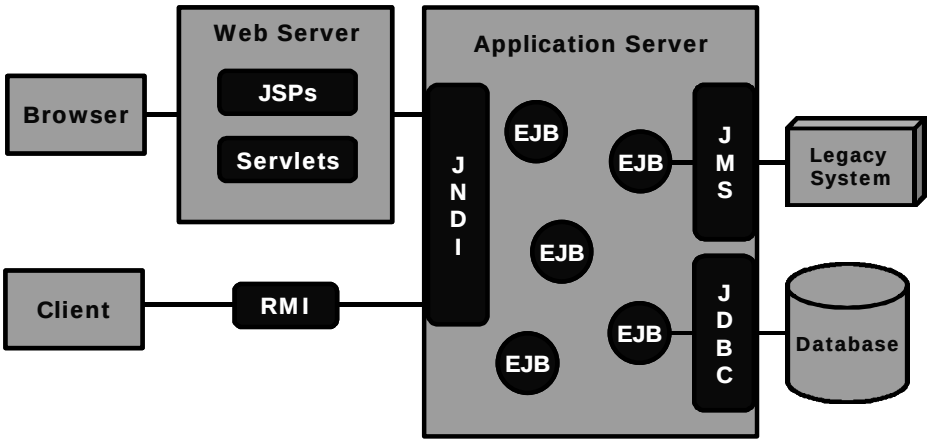

Figure 1: The J2EE Platform

EJB containers are normally implemented as one of the services provided by an application server. A typical application server supports web-based applications, via a web-server and can communicate directly with Java clients using the Remote Method Invocation (RMI) protocol. Databases can be accessed using the Java Database Connectivity (JDBC) API or native database drivers. The Java Messaging Service (JMS) provides a means to communicate with non-Java legacy applications.

Java Server Pages (JSP)

JSPs provide a mechanism for web page designers to create dynamic web content by writing a relatively small amount of Java code. JSPs remove the need for tedious hand-coding of HTML pages using lots of Java print statements. A JSP consists of HTML code interspersed with Java statements. When the page is requested by a client, the server executes the Java code and returns the generated HTML pages to the client.

Servlets

A Java servlet is a small Java program that extends the functionality of a web server in much the same way that an applet extends the functionality of a web browser. Servlets are automatically executed whenever a browser requests the URL associated with the servlet. Servlets and CGI scripts serve the same purpose but servelets have a major advantage. CGI scripts are executed in dedicated server processes. In contrast, servlets are executed as a separate thread within the web server. This means that servlets offer much greater scalability than CGI scripts.

Java Naming and Directory Interface (JNDI)

As its name suggests, the JNDI API is used to access naming and directory services. It provides a consistent model for accessing and using

resources such as DNS, LDAP, local file systems and application server objects. JNDI is intended to facilitate the portability of applications.

Enterprise Java Beans (EJBs)

EJBs are components which are executed inside a "container". EJBs provide a framework for developing and deploying distributed business logic. The J2EE specification defines how an EJB should interact with its container. The container provides generic services such as security, transaction management, resource pooling and fault tolerance. EJBs implement the business logic of the application. This approach frees the application developers from concerns such as transaction management and security, allowing them to concentrate on developing just the logic related to the problem domain.

The J2EE specification defines three categories of EJB:

- **Stateless Session Bean**. This type of EJB can provide services to multiple clients. Stateless Session Beans are volatile and do not survive container crashes. A currency conversion service is a typical example of how a Stateless Session Bean might be used.
- **Stateful Session Bean**. This type of EJB maintains the state of a single client. Stateful Session Beans are also volatile and do not survive container crashes. A shopping cart is a typical example of how a Stateful Session Bean might be used.
- **Entity Bean**. This type of EJB stores the state of a business object. Entity Beans are non-volatile and will survives container crashes. A customer account is a typical example of how an Entity EJB might be used.

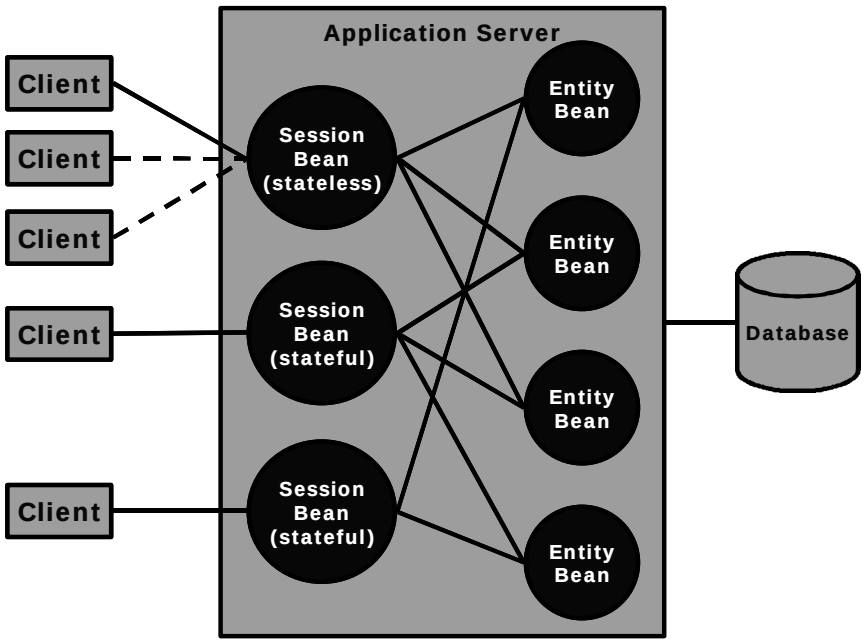

Figure 2: Relationships Between EJB Instances

Figure 2 illustrates the relationships between Session Beans, Entity Beans and the clients that use them. As the diagram shows, Stateless Sessions Beans can be used by a number of clients. The clients use the Session Bean on a message by message basis. In contrast, Stateful Session Beans can only be used by a single client during a session. Both types of Session Bean can make references to multiple Entity Beans. Conversely, each unique instance of an Entity Bean has the potential to be referenced by more than one Session Bean.

Out of necessity, this has been a brief introduction to the J2EE platform. A more in-depth discussion is beyond the scope of this paper. However, there is a wealth of information about the J2EE platform available at Sun Microsystems web site [2]. For a more detailed but concise overview see [3].

J2EE and People

From a people-perspective, it can be argued that the J2EE platform represents a "clash of cultures". The Java platform has evolved from the world of software engineering and object oriented programming. Originally, Java was designed as a means to develop software to be embedded in hardware systems. One of the goals of Java's designers was to improve on the object oriented programming language C++.

The J2EE platform on the other hand is clearly targeted at the developers of information systems. Traditionally, this group has not used object oriented programming languages to develop applications.

These two tribes of developers frequently have very different perspectives of the software development process. They employ different techniques, have different skill sets and different traditions. Some observers have named this clash of cultures the "object-data divide" [4].

The Object Oriented Perspective

At times, there is so much hype surrounding object oriented programming (OOP) that it is easy to forget how it started. OOP was originally developed as an alternative to the traditional programming model. The traditional model is based on the underlying computer hardware that executes a program. The hardware architecture's rigid separation of storage and processing functions are reflected in the separation of data and logic in a program.g

The separation of data and logic means that programming problems must be decomposed into separate "data" and "logic" elements before they can be

solved. While this now seems quite natural for many software developers, there are some categories of problem that can be very difficult to solve using this approach.

OOP officially started in Norway in 1967 with the development of the Simula language [5]. Simula is a simulation language designed for writing software simulations of the real-world. Simulations are one of the classes of problem that can be very difficult to solve using the traditional approach.

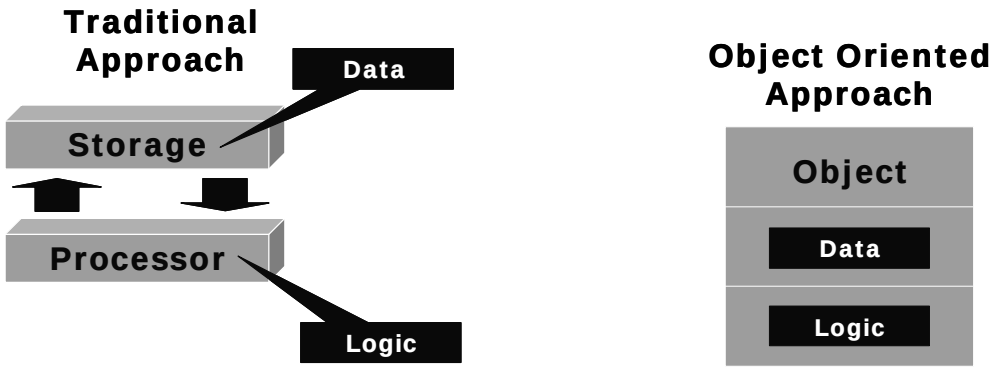

Figure 3: The Traditional Programming and Object Oriented Programming Models

Simula introduced a new programming model based on software "objects" rather than the underlying computer hardware. Simula's software objects could easily be configured to mimic the various components of whatever was being simulated. Simula's objects also employed the principle of "encapsulation" to hide an object's data values from the outside world. Only logic which forms part of the object may change the values of its "private" data.

Smalltalk, developed in the 1970's, was actually the first language to be called "object oriented". Smalltalk built on the approach pioneered by Simula. Although other object oriented languages, such as C++, followed Smalltalk during the 1980's, OOP did not really become "mainstream" until the early 1990's.

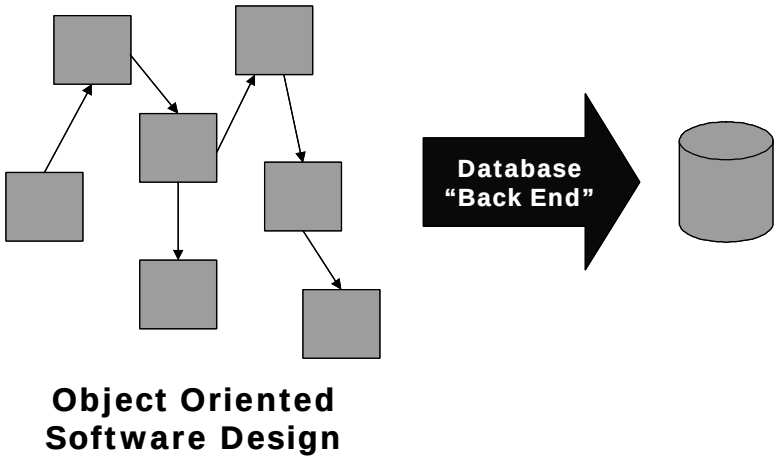

Figure 4: Object Oriented Software Design

An object oriented software design consists of a number of related object classes which "collaborate" to deliver the required application functionality. From the perspective of an object oriented developer, the object oriented design is the center piece of the application. Frequently, some of the classes in the design need to be "persistent classes". The data encapsulated inside a persistence class must survive system crashes and restarts. In other words, persistent classes must be stored using some non-volatile storage medium such as a file or database.

A few years ago, many object oriented developers believed that object oriented databases would eventually replace relational databases. They looked forward to the time when object oriented programming languages combined with object oriented databases, would provide a seamless mechanism for making classes persistent. However, object oriented databases have not grabbed the mind or market share that was originally hoped for by the object oriented community.

This, coupled with the amount of legacy data already stored in relational databases, means that it is common for persistent classes to be stored in a relational database. However, the attention of object oriented developers remains fixed on the object oriented software design with the result that they often regard the relational database as a relatively unimportant "back end" to their software.

The Information Systems Perspective

The history of information systems development is not as well charted as that of OOP. Development of information systems has gone through a number of evolutionary stages starting in the "dark ages" with assembly language and punched cards. As a rule, improvements in programming languages and data management techniques have taken place independently of each other. Starting with assembly language, traditional programming languages have evolved through COBOL to Fourth Generation Languages (4GLs). Starting with punched cards, data management has evolved through file systems, hierarchical and network databases finally arriving at relational databases.

As a rule, the evolution of information systems has been opportunistic and driven by practitioners rather than academics. In spite of this, theoretical foundations for the study of information systems have been proposed [6]. One perspective that has appeal proposes that:

> *information systems are used to represent the structure and behaviour of other systems. They are intended to be the basis for coordinated action in some social system, for example an organisation.* [7]

To paraphrase this, an information system removes the requirement to directly observe what is happening in the real world. Information systems achieve this by representing the states of things in the real world as tables stored in a database.

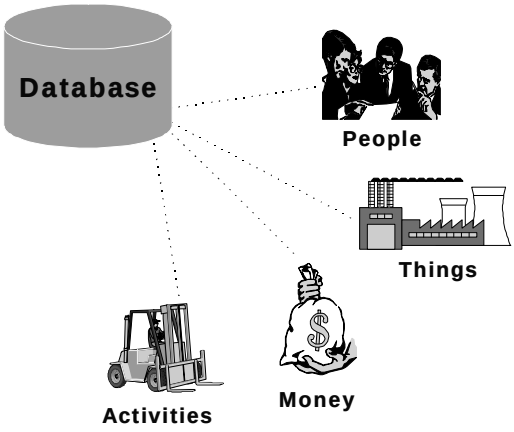

Figure 5: Information System Capture the State of the Real World

From the perspective of an information systems developer, the database design is the center piece of an application. The database is frequently seen as the "foundation" on which the application programs rest. This view is reinforced by the fact that data requirements of an application, have a tendency to be more stable than the processing requirements. This, coupled with the independent evolution of programming languages and data management mentioned above, has meant that some information systems have undergone a number of incarnations. Often, the underlying database has remained essentially unchanged, while the programs which access the data have been redeveloped. The result is that information systems developers have a tendency to regard the programs that access a database as a an interchangeable "front end" to the database.

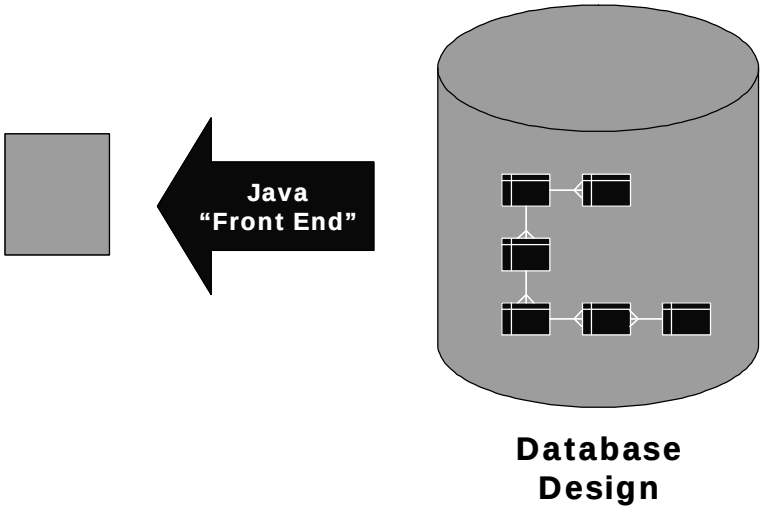

Figure 6: Database Design

Six Blind Men and an Elephant

In isolation, the object oriented and information systems perspectives are perfectly valid and have served both communities well for a number of years. However, problems can arise when the two groups are brought together. If one group or the other starts to advocate that their perspective is the “only valid view” then conflict is bound to follow. Conflict of this nature is not unknown to the religions of the world and is aptly illustrated by this Jain story [8].

> *There were six blind men who lived in a village. One day, an elephant wandered into the village. The blind men had never encountered an elephant before, so they decided to go and touch the elephant to see what it was like. As they approached the elephant and touched it each man described his impressions.*
>
> *"An elephant is like a pillar," said the first man who touched a leg.*
>
> *"Oh, no! It is like a rope," said the second man who touched the tail.*
>
> *"More like a thick branch of a tree," said the third man who touched the trunk.*
>
> *"I would have said a large fan" insisted the fourth man who touched an ear.*
>
> *"No! No! A huge wall," said the fifth man who touched the belly.*
>
> *"I would say a pipe," Said the sixth man who touched a tusk.*
>
> *They began to argue about the elephant. Each man insisted that he was right. Fortunately, a wise man was passing by and he heard the disagreement. The wise man stopped and asked them what they were arguing about. "We cannot agree on what an elephant is like", they said. Each man described his impression of the elephant. The wise man laughed and explained that they were all correct.*
>
> *“The reason that you each describe the elephant differently, is because each of you touched a different part of the elephant. In fact, the elephant has all those features that you described."*

The Conceptual Model Perspective

As far as the J2EE platform is concerned, placing too much emphasis on the software or database design is analogous to the blind men describing the elephant. Neither view is incorrect but they both describe a single perspective of an application.

Ironically, given the object oriented foundations of J2EE, we are dealing with the same separation of data and logic that prompted the development of Simula! However, this time the separation is not so obvious because of the strong similarities between object oriented software models and database models.

The problem is best illustrated by a simple example. An object oriented software design for a bank might include an object class `BankAccount` which represents a customer's bank account. A database model might include a table `BANK ACCOUNT` which also represents a customer's bank account. Because the same abstraction is used for both the object class and the table it is easy to assume that the two are roughly equivalent. However, a careful comparison reveals that an object class and a database table are in fact, quite different things.

The `BANK ACCOUNT` table has a row (instance) for every account opened by the bank. If there are 100,000 accounts there will be 100,000 rows in the table. As we have seen, rows in the `BANK ACCOUNT` table represent things in the real world and they must retain their values when the system crashes or is restarted. If we exclude triggers and stored procedures, `BANK ACCOUNT` tables are passive and do not implement any business logic.

In contrast, it is highly unlikely that there will be an instance of the `BankAccount` class for every account opened at the bank. This is because swoftware objects are stored in main memory (as opposed to disk). Creating 100,000 instances of the `BankAccount` class would quickly saturate the available memory.

In addition to this, software objects do not need to retain their state when the system crashes or is restarted. Their state can simply be reloaded from the database. Unlike tables, software objects are not passive. Their methods implement the logic of the application.

Once these fundamental differences are understood, it is easy to see that an object oriented software design is unlikely to provide a good basis for a database design. As we shall see, the converse is also true, a database design does not provide a good basis for a software design based on EJBs.

Figure 7: Software and Database Models Are Not a Good Basis for Designing Each Other

What is required is a model that provides the complete picture of an application. Such a model needs to place equal emphasis on both the data and processing requirements of the application. Such a model is frequently referred to as a "conceptual model".

Conceptual models are based on the same sort of abstraction as software and database models but with a number of important differences:

- "Object types" rather than software classes or database tables are used to represent problem domain concepts.
- In contrast to database tables, object types are not passive. They can represent the behaviour of something in the real-world.
- In contrast to software classes, object types are not transient, they can represent the persistent state of something in the real world.
- In contrast to database tables, object types do not represent all real world instances. Neither do object types represent a restricted subset of instances in the same way that software classes do. Instead, object types represent "typical" instances of things in the real world.
- Finally, a conceptual model provides a good basis for both software and database designs.

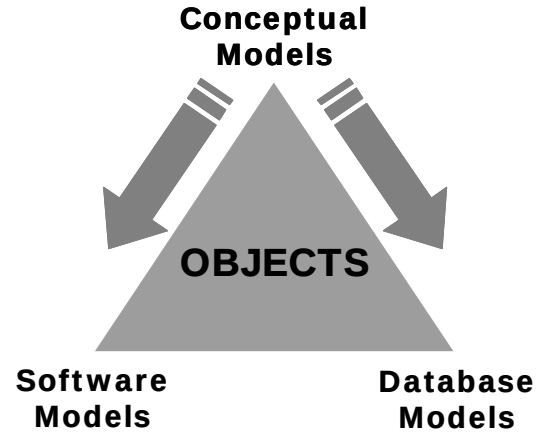

Figure 8: Conceptual Models as the Basis For Software and Database Designs

Conceptual models can also provide the foundation for better communication between the object oriented and information systems developers. Even better, if both groups participate in developing the conceptual model, a shared understanding is forged between them.

Model	Abstraction	Instances	Volatility	Logic
Software	Class/Object	As required during execution	Transient	Application Logic
Database	Table/Row	All real world objects	Persistent	No Logic
Conceptual	Type/Object	Typical real world objects	Real world state	Real world behaviour

Table 1: Comparison of the Three Types of Model

The key differences between object oriented software designs, database designs and conceptual models are summarized in Table 1 above.

Online Auction Example

Many of the ideas presented above are somewhat abstract in nature. A short, concrete example is offered to illustration the benefits of conceptual modelling. The example is based on a web site that conducts on-line auctions. Parts of the site can be viewed by any casual `Browser`. If a `Browser` wants to bid in an auction they must `Register` to become a `Member`. A `Member` can `Logon` to the site and make `Bids`. `Sellers` are `Members` who post something to sell at the web site. A `Seller` can `Add Auction Items` and `Open Auctions`. A `Buyer` is a `Member` who has made the highest bid when the auction is `Closed`. A winning bid must be above the reserve price set by the `Seller`. Auctions are `Closed` automatically on the date and time set by the `Seller`. The `Buyer` must `Make Payments` for the `Auction Item` that they have purchased. One of the tasks performed by the back office `Staff` is to `Maintain Categories` of `Auction Items`.

These requirements are summarized in the Use Case Diagram below.

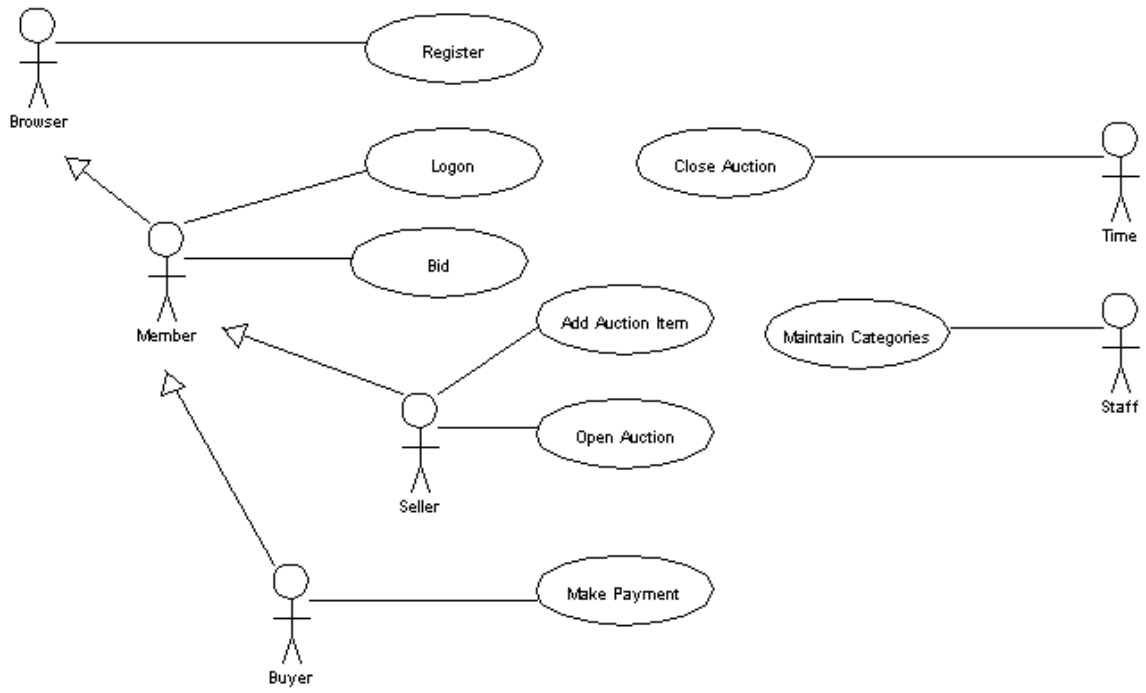

Figure 9: Online Auction Use Cases

The database design for the online auction site is shown in the data model diagram below.

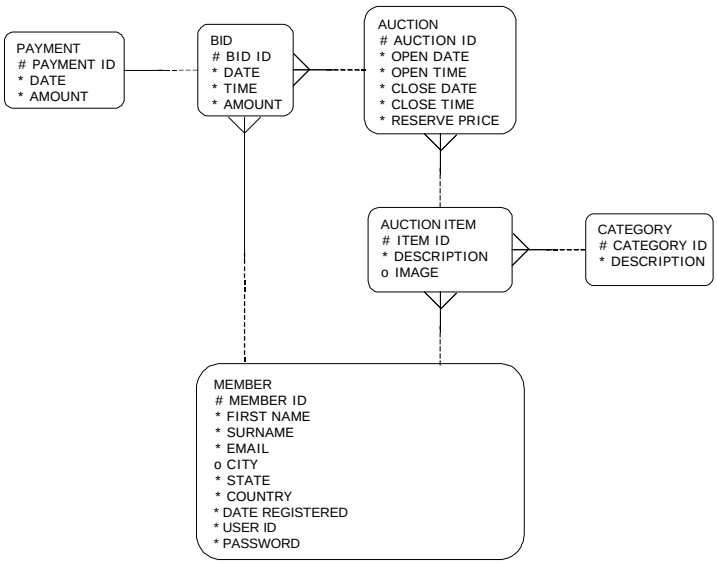

Figure 10: Online Auction Data Model

It is tempting to take the data model above and use it as the basis for creating a software design for the Entity Beans. A design based on the data model would include an Entity Bean for each table. The use of the term "Entity Bean" in the J2EE specification even suggests that this is the correct thing to do. Entities are a familiar concept to data modelers which automatically suggest the use of tables.

However, this approach would suffer from a number of disadvantages:

- The client will need to make many "fine-grained" remote references to the Entity Beans. The remote references will be taking place across the network and will result in increased network traffic.
- If an Entity Bean contains a reference to a second Entity EJB the reference will be implemented as a remote reference. This is done to allow for the possibility that Entity Beans can reside in different containers but still reference one another. Remote references incur a significant performance overhead when compared to normal object references.
- The software design would become dependent on the database schema. Changes to the database schema will require parallel changes to the software design. This represents a step back from the data independence that information systems developers have come to expect.

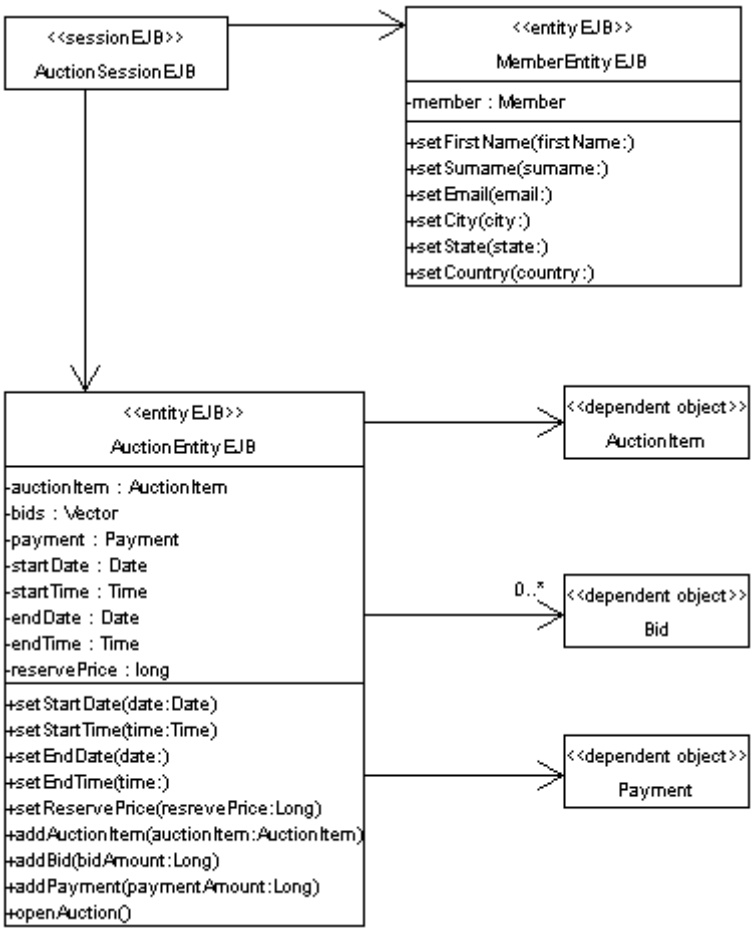

Figure 11: Online Auction Software Design

The disadvantages listed above, suggest that the database model does not provide a good basis for an Entity Bean design. A better approach is to incorporate Entity Beans into the design as "coarse-grained business components" which are shared between clients. The role of the Entity Beans is to encapsulate the business and data access logic associated with the persistent classes. The result can be regarded as a business-oriented interface to the underlying database [9] and [10].

Applying this approach to the online auction application leads to the potential design shown in Figure 11. `AuctionEntityEJB` is mapped to the `AUCTION`, `AUCTION ITEM`, `BID` and `PAYMENT` tables in the database. `AuctionItem`, `Bid` and `Payment` are "dependent objects". A dependent object does not exist in its own right. Instead, it is dependent on and has its life cycle managed by an Entity Bean [10]. In this case, `AuctionEntityEJB` will create new instances of `AuctionItem`, `Bid` and `Payment` as they are required to store the data retrieved from the `AUCTION ITEM`, `BID` and `PAYMENT` tables in the database.

The design of `MemberEntityEJB` illustrates that the approach described above is simply a design guideline rather than a rigid formula to

be doggedly applied. In this case, MemberEntityEJB has been mapped to just a single table (MEMBER) in the database.

So far, we have not discussed the role of Session Beans in the software design. In this example, AuctionSessionEJB implements a session façade [10]. The purpose of a session façade is to hide the details of accessing Entity Beans from the client. A session façade also handles relationships between Entity Beans, removing the need for the Entity Beans to reference each other. This avoids costly remote references between Entity Beans. The result is improved performance and a simple, service-oriented interface to the client.

However, care must be taken that session façades do not become excessively "bloated" by including too much of the application logic. This approach would be a step backwards to the "monolithic" style of application design. This potential danger can best be avoided by ensuring that application logic is well distributed between the Session and Entity Beans. In the example, AuctionEntityEJB includes operations such as addBid(bidAmount:long) and openAuction() which implement business logic. These operations are in addition to the simpler data "setting" and "getting" operations which simply store and retrieve data values. They provide the domain logic of the application.

Although not shown in the example, AuctionSessionEJB will also include operations such as addBid(bidAmount:long) and openAuction(). However, these operations will simply "delegate" to AuctionEntityEJB.

An important thing to note about the design shown in Figure 11 is that it would definitely not provide a good basis for a database design. If the database consisted of just two tables, MEMBER and AUCTION, it would not be properly normalised. This would most likely lead to performance problems and make the database difficult to modify at a later date.

The conceptual model for the online auction is shown in Figure 12 below. The model represents problem domain concepts as object types. Object types include real world state and real world behavior. They represent "typical" instances objects in the real world.

In the example, Person, Membership, AuctionItem, AuctionCategory, Auction and Payment have state but no behaviour. Member, Seller and Buyer have behaviour but no state. Staff and Bid have both state and behaviour. PersonRole is an abstract concept which has been included to help structure the model.

In this example, the conceptual model has been presented as a UML Class Diagram but this is not the only option available to conceptual modellers. Techniques such as Object Role Modelling (ORM) [11] and The IDEF5 Ontology Capture Method [12] could equally well have be used to develop

the conceptual model. However, the there are significant advantages to using the standard UML notation to describe all three models. The only danger is that the common notation will lead to misunderstandings and arguments about whether the diagrams describe an elephant's ear, an elephant's trunk or the entire elephant!

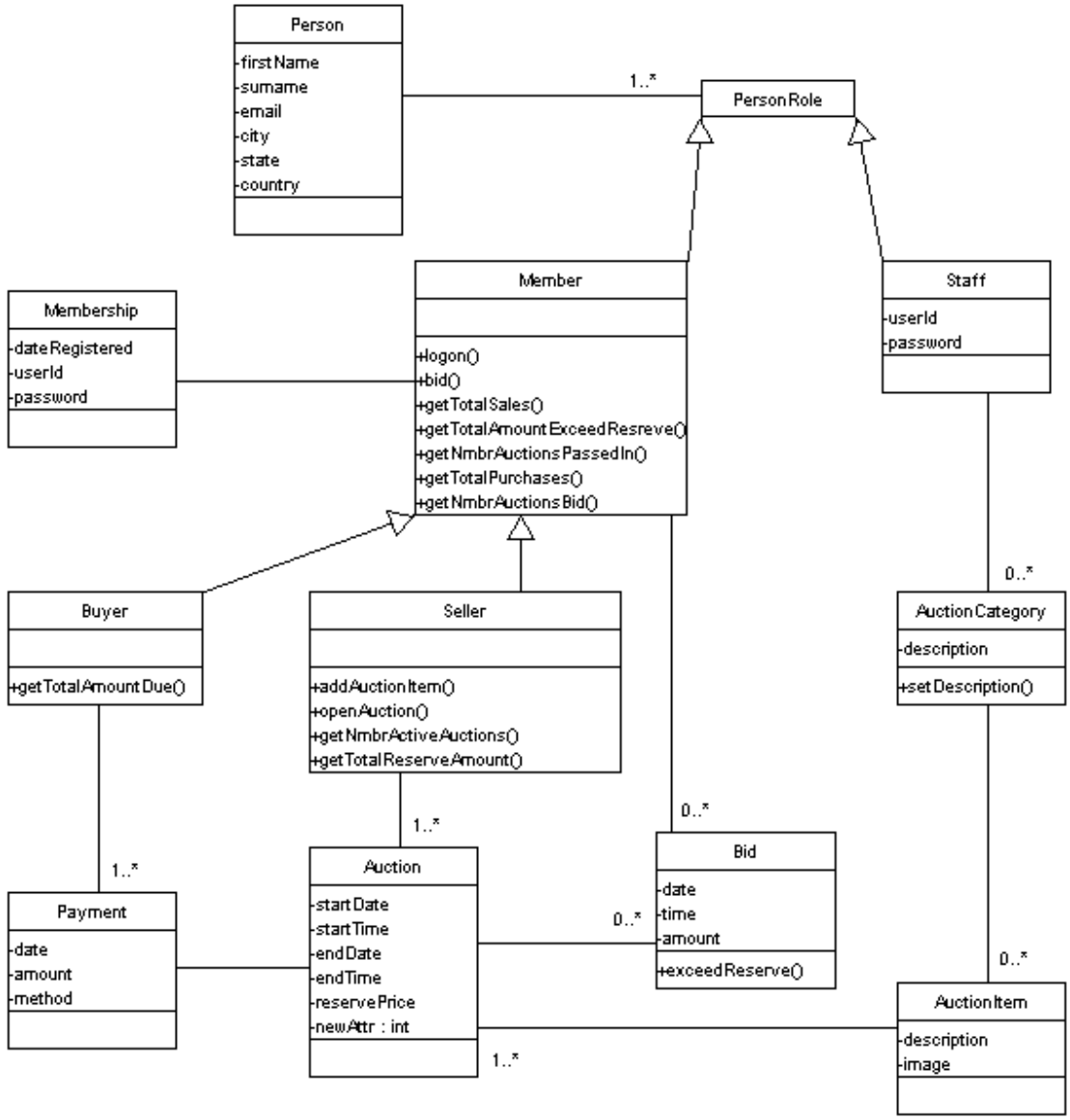

Figure 12: Online Auction Conceptual Model

---

[1] Pour, G., "Enterprise Java Beans 101: Sever-Side Components", *Software Development*, April 2000.

[2] See Sun's web site at http://java.sun.com

[3] Gould, S., "Develop n-tier applications using J2EE", *Java World*, December 2000.

[4] Ambler, S., "Crossing the Object-Data Divide", *Software Development*, March 2000.

[5] See Kristen Nygaard's web site http://www.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html

[6] Weber, R. “The Information Systems Discipline: The Need for and Nature of a Foundational Core”, *Proceeding of the Information Systems Foundations Workshop on Ontology, Semiotics and Practice*, 1999.

[7] Shanks, G., “Semiotic Approach to Understanding Representation in Information Systems”, *Proceeding of the Information Systems Foundations Workshop on Ontology, Semiotics and Practice*, 1999.

[8] See the *Jain World* website at http://www.jainworld.com/literature/story25.htm

[9] Larman, C., “Enterprise JavaBeans 201: The Aggregate Entity Pattern”, *Software Development*, April 2000.

[10] Alur, D., Crupi, J., Malks, D., *Core J2EE Patterns: Best Practices and Design Strategies*, Sun Microsystems Press, 2001.

[11] Haplin, T. A., Conceptual Modelling and Relational Database Design, Second Edition, Prentice Hall, 1995.

[12] *The IDEF 5 Method Report*, Knowledge Based Systems, Inc, 1994.