SELECTED FEATURES OF OBJECT-ORIENTED DATABASE TECHNOLOGIES FOR STATIC-TYPED BUILDING PRODUCT MODEL DEVELOPMENT

Chris Price, Scientist
Building Research Association of New Zealand Inc.
Private Bag 50908, Porirua.
Wellington, New Zealand
branzcgp@branz.org.nz

Abstract

This paper presents a methodology comprising a selection of established and new object-oriented database technologies. This methodology is presented in the context of the development of static-typed class libraries for Building Product Model (BPM) development and delivery.

Having established the environment in which to develop BPMs, some important issues are discussed: the importance of correct object identification in terms of the possibilities for concurrent and distributed computing; how inter-object relationships can be safely and efficiently implemented and finally, tthe ways that object databases or persistency mechanisms relate to the application portability and the operating system.

1. INTRODUCTION

Strategies for the development and delivery of building product models (BPMs) encompass many different areas of active research, modelling efforts and diverse technologies and methodologies.

The author presents in this paper a selection of established and new technologies associated with the development of software components. It is suggested here that the use of common models within software components is a viable path to the development and delivery of BPMs.

A debate on strategies for BPM development and delivery involves all aspects of software engineering, marketplace acceptance, trends in existing software design and development, successes and failures of past standardisation attempts and existing data exchange practices. Some of these points are touched on, in this paper.

This paper is about technologies and methodologies associated with the software component path and does not enter fully into the overall debate. However, to make some justification for the chosen path, here are four important aspects concerning the full process from development to industry use are listed here:

1. The development of sophisticated models of "the building life-cycle" that are descriptions of the built environment objects, and the processes of information transformation.
2. A means to integrate incompatible models of the same collection of objects.
3. Standardisation to enable common data sharing.
4. Implementation, delivery **and use** of design tools, data transfer tools and, at the high-end, concurrent design environments.

With these aspects in mind, the following list categorises possible paths towards BPM development and delivery. The paths range from schema development to efforts to implement software components. Standardisation ranges from international co-operation to single in-house proprietary development. Proprietary development is included because it is the only successful path so far for building models that are widely used within the building industry.

- *The creation of neutral data exchange mechanisms for existing software.* The methodologies and technologies have been established with the STEP DIS (Standard for the Exchange of Product Data, Draft International Standard), (STEP, 1991). However current research work is focusing on greater concurrent engineering issues than the STEP architecture allows, for example Hosking et al (1994) and Amor (1994).
- *Proprietary CADD (computer-aided draughting and design) systems with third-party add-ons.* This could be from major upgrades from existing established CADD vendors, e.g. AutoCad. Totally new "BPM CADD" could appear from current development within research groups, or perhaps an integrated group of systems each dealing with different BPM design and data processing. An example is the Combine integrated architecture (Augenbroe, 1992).
- *Software components, where the BPMs are developed and implemented and presented as program building blocks from which many diverse CADD systems can be created.* An initial example is Cadkey (1994).

Building descriptions in CADD systems tied to other applications, such as bill-of-material or specification report creation, are the current major example of computer-integrated construction used by the building industry. Today there are a few specialised building CADD systems, and a number of add-ons to general CADD systems. Almost all are proprietary. Examples are listed in major CADD vendors software catalogues (e.g. Intergraph, 1994).

Data exchange seems not to be a concern amongst these vendors, as they compete for market share. A survival-of-the-fittest evolution of BPM applications could be a probable path to BPM usage. This is what is occurring in the PC market today for applications such as word-processors and spreadsheets, which are general

tools. The word-processor market has become dominated by a few very comprehensive products, whose complex data structures are proprietary. An example of this occurring for more specialised topics is harder to find. The surface rendering models in Pixar Corporation's RenderMan (Pixar, 1988) and Silicon Graphics Inc's Open GL specification (Silicon Graphics, 1992) are the most sophisticated, well-known examples. However, these are still generic, not the specialised models of real-world objects which is a BPM.

The neutral data modelling and processing paths cover most of the current international, national and industry projects. The aim of neutral data exchange is to be able to transfer structured data, by the description of the data structures and their meaning within any CADD system or any other application, in a format that can be unambiguously interpreted by computer processing tools. This path is required in a software industry that builds independently designed systems that are not created from any common building blocks. In the STEP methodology, neutral data exchange occurs through the specification of an application protocol for a domain (or topic), to which CADD vendors must conform.

The software component path is quite different to neutral data exchange in that it relies on the software industry using common software components to build future CADD systems, etc. As will be explained in the methodology section, it is more about sharing (reuse) of common design rather than enabling unambiguous data exchange between diverse schemas (data structure). It does not solve the neutral data exchange requirements. Where possible (i.e., a willingness to use new generations of CADD) the use of common design in future CADD systems avoids the need for neutral data exchange. If a software component industry does mature, perhaps BPM research and standardisation should be more focused on how common design can be promoted. This path has a few supporters amongst BPM researchers (Price, 1993a).

2. THE METHODOLOGIES

2.1 OBJECT VERSUS DATA SPECIFICATION

The term "development and delivery" used in this paper is a path to BPM use through the creation of BPM software components incorporating standardised BPM designs (Price, 1994). It is the combination of modelling and implementation (or programming).

This route to BPMs is quite different from the methodology central to the established neutral data exchange. Neutral data exchange uses information modelling to communicate the data structures and their meaning between different CADD systems and databases. Part 11 of STEP describes the Express data specification or information modelling language. The current STEP Express (version 1.0) processing tools available are listed in Wilson (1993). The ability to

unambiguously describe all the meaning behind the data structures and their inter-relationships within CADD systems is a major challenge.

The author chose an alternative path to that of neutral data exchange (developing software components) after initial BPM development work. This consisted of searching through most of the analysis and design methodologies and technologies that appear in data modelling and programming, and object-oriented and relational database design. Various technologies and notations have been tried in the creation of a BPM for timber-framed houses (Price, 1993b). Examples are: NIAM (Nijssen and Halpin 1989), Object-oriented Modelling and Design (Rambaugh et al, 1992), Express data modelling (Wilson, et al, 1993), the C++ programming language (Stroustrop, 1990) and some early class libraries, plus one type of object-oriented (OO) database (Godard and Simmel, 1994).

Over a period of three years the Author participated in e-mail discussions on STEP technologies and the associated information modelling methodology. It became quite clear in this period that the techniques (e.g. application protocol integration, the Express language, the emphasis on classifying without a concern for implementation) were not adequate for the quite ambitious integration aims of STEP (Email,1993).

The software component path is about the creation of common application design rather than an attempt to be able to describe the complexities inherent in various CADD databases. The development of common designs includes the integration of data structures, but avoids having to completely transfer the underlying meanings and relationships amongst the data schema. In fact the object-oriented programming idea of hiding details is used.

We as BPM developers can offer potential BPM application developers more than just a data specification; we can offer software components for the BPM application developer to use. This is the provision of code that can manipulate the data, as well as view it. New data definitions require code to manipulate them.

What proportion of the overall project is the data definition stage? The authors estimate is about five per cent. The remaining effort is in the editing and manipulation algorithms on the data. BPM class libraries consisting of objects that can be stored in a database provide far more to the specialised CADD developer than just a data specification. If CADD vendors are to use a neutral data exchange specification, they have to make use of external data processing tools to link the data specification to their own CADD system's internal data structure.

A BPM in this path will be a sophisticated container class library. A class library is a definition of a set of objects, their structure and behaviour. A container class defines the characteristics of a collection of objects. The container can hold any type of object. The container class defines what can be done with the collection of objects (insertion, deletion, ordering, lookup, navigation, views, queries, etc).

Included within such a BPM class library will be different representations of the same set of objects. To link them, neutral data exchange technologies such as multi-schema mapping (Amor, 1994) will be incorporated. The term "engine" can be taken from graphics, virtual reality and game software components, and applied to a BPM class library that includes all these features. Figure 1 illustrates the similarity between neutral data exchange and the internals of a BPM engine.

As of 1994, a few CADD vendors (for example Cadkey 1994) have introduced class library interfaces to their CADD systems. These define the data structures, interfaces and database access aspects for application developers. The reuse of these class libraries, the integration of a number of different systems, and their specialisation to building applications will be central to BPM class library development. Application programmer interfaces (APIs) to these systems have been around for a number of years, but the introduction of class libraries heralds the start of better software component development.

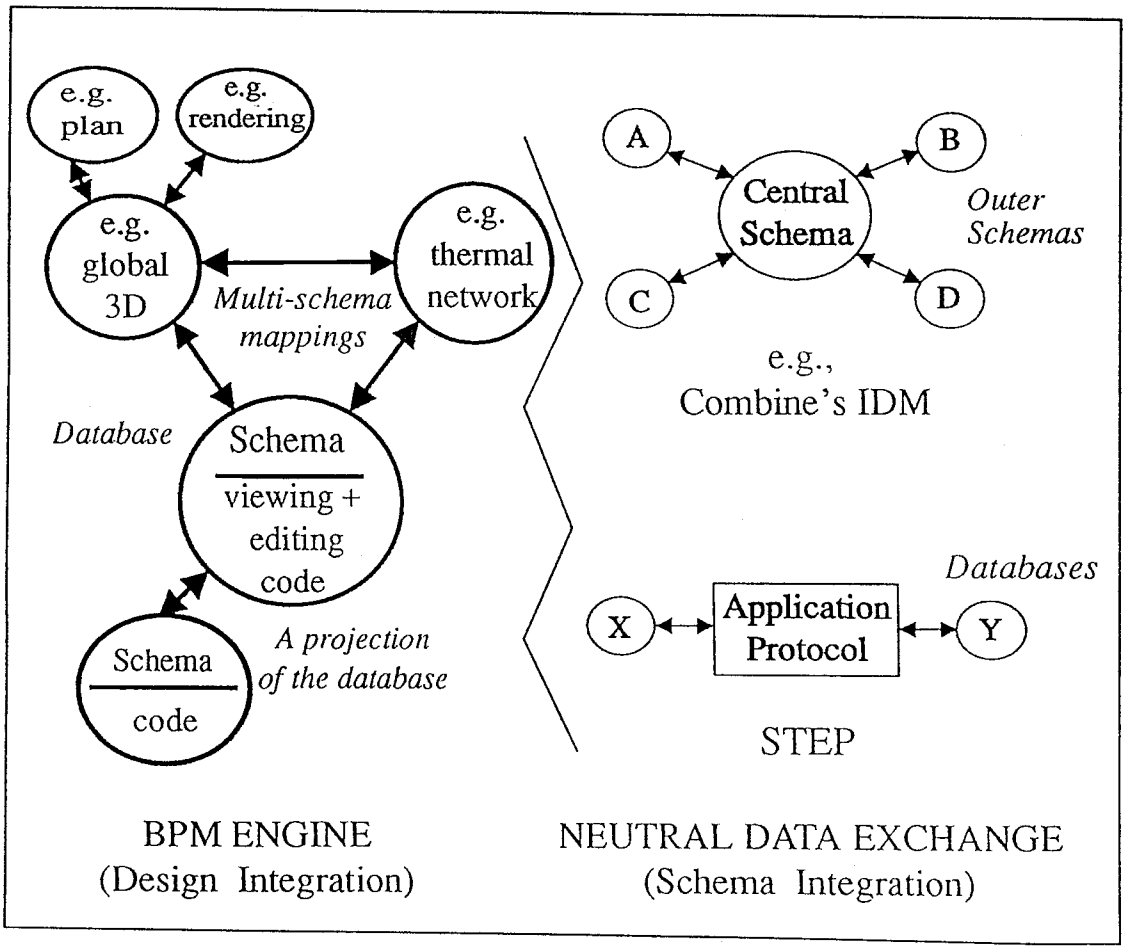

Figure 1:Neutral data exchange and a BPM engine.

It is necessary to standardise what are the capabilities of an object without being specific as to its structure. Class library technology provides this in the form of the definition interface that (correctly written) hides arbitrary implementation choices relating to the object's structure and behaviour. This basic object-oriented feature is missing within current data specification (e.g. Express) languages.

The information-hiding interface becomes necessary as the collection of interrelated objects becomes more complex, and the choice of how to implement the object's internals, and the links between objects, grows rapidly. For instance, the domain of solid representation is a world of vertices, edges, and faces, and various transform and jointing operations to represent solid objects. Requicha (1980) lists eight general categories of rigid solid representations (such as constructive solid geometry, sweep surfaces and boundary representation). Boundary representations (the description of an object in terms of its enclosing boundary) have particular relevance to the author's current directions in BPM development. Again, several ways of implementing representations exist, as outlined by Karasick (1989). However, one can take these different implementations and assign common methods or defined behaviours. Requicha makes an initial analysis of applicable solid behaviour capable for each rigid solid representation scheme. More recent examples of common methods and behaviours are the regular CADD evaluation articles that have desirable-feature tables (Forcade, 1993). The common methods are the listed features on which the CADD systems are compared.

2.2 BPM EVOLUTION

Building product models are far from mature. Existing models from the several BPM projects developed internationally are on different aspects of buildings, and the basic model schemas defined have diverse designs- for example the RATAS project (Bjork, 1989), the IDM (integrated data model) schema within the Combine project (Augenbroe, 1992), and the various STEP application protocols (STEP, 1991). The schemas may have to be developed to a stage that can differentiate all the different details and arrangements of objects inherent in various building system/subsystem designs. As well the scope is not defined, as the level of detail for the over-generalised term "building product model" is undefined.

The STEP methodology is not appropriate for developing ideas; rather, experts are gathered together to standardise models on what is existing technology. This is seen in the guidelines for application protocol development protocols (Palmer, et al, 1992), and the use of domain experts to summarise rather than design. As BPMs are refined, submissions should be made to standardising groups in the form of conceptual schemas, written in Express. Before that there is a need for close analysis, design and implementation of BPMs. The ideas have to be put into practice. As feed-back occurs on the implementation consequences of BPM schema design, changes will occur.

To begin developing a BPM for timber-framed houses, the author first reviewed the extensive literature on this subject. This included design solutions (style, solutions, manufacturers' catalogues), presentation systems (classification systems, terms, graphic detailing), principles and requirements (standards, building controls, checklists), and processes (Price, 1993b). Of particular interest were codes of practice that have been computerised by BRANZ in the past (Dechapunya et al, 1990), (Hosking et al, 1992).

Amongst this complexity, the author chose a line of modelling that focused on geometry associated with grids, and the associated systems of building elements and assemblies. A hierarchy of classes for building is under development based on grid systems developed by Turner (1989). This is just the skeleton for a sophisticated class library for a BPM subset.

2.3 THE BPM DEVELOPMENT ENVIRONMENT

The BPM class library development environment involves the integration of analysis, design and implementation. The choices of technologies is centred on where the models are developed and the effect of design iterations on program coding. The two aspects here are the degree of computer-assisted modelling tools and whether the model development occurs within the database environment. For example, a large portion of the relational database feature list relates to data model manipulation. Most OO databases aim to provide the same features offered by relational database environments.

2.3.1 Computer-aided Software Engineering Tools

Computer-aided software engineering tools (CASE) are not used yet. The creation of concept diagrams, class/instance relationships and the subsequent code maintenance are all done manually.

With respect to BPM development, there are two main requirements for the application of CASE:

1. To automate the conversion of diagrams and concepts to code, and to manage changing definitions of classes as the design is changed and,
2. To unify many existing diverse and seemingly incompatible building models extracted from existing software (Amor et al, 1994).

So far during the development of the basic class library, the maintenance task arising from changing a class definition is the necessary alteration of any related class function (code associated with a class). An example of such a function (or class method) is the initialisation of objects (instances of the class). But by far the largest development tasks are the creation of methods that implement object manipulation algorithms. This points again to the need for complete class libraries rather than just data specifications.

The second requirement for CASE is not used here because of the size and current focus of modelling within the project. The project is at a stage where a particular representation of a building is being developed and demonstrated by developing a BPM database and a Windows-based application to enter and manipulate the BPM objects. The project is specialising of building representations, rather than generalising and integrating existing building models.

2.3.2 Database Technology

With regard to the database technology to be used, there is a choice between traditional relational databases and the new object-oriented databases.

Relational database technology is not used. There is the standard problem of "impedance mismatch" in fitting objects into tables (relations). However, the overwhelming factor is the relational database's mechanism to pick an object from a collection by its value. This is an inherently inefficient operation compared to databases that deal with object identity. Object identity expands the use of pointers so vital in CADD (see Section 3.2), to a concurrent engineering environment where unique object identification is used to retrieve an object from any database that has groupings of objects necessary for versioning and shared ownership.

One feature that is associated with relational databases is the ad hoc query feature. A query is a request for data that comply with the user's requirements. The "ad hoc" prefix indicates that there is a freedom to formulate complex queries by the combination of requirements. The successful SQL (standard query language) standard (SQL, 1984) provides a common query language.

A container class will define procedures for accessing the data. These can provide ad hoc querying to a limited extent. A few OO databases provide query facilities based on SQL.

The chosen object-oriented database was not a fully featured object-oriented database environment (features as listed in commercial OO database comparisons), but a "data server". A data server category of OO database provides just the basic mechanisms to implement your own persistent data environment (Godard and Simmel, 1994). It does not include tools to manage the data model. A persistent data environment simply means that the data is stored and so lasts longer than the lifetime of the execution of a program.

Some important features of existing OO databases with regard to BPM class library implementaion are discussed in Section 2.5.

2.4 THE OBJECT LANGUAGE MODEL

The established object-oriented programming technologies offer solutions to handle program complexity and the ravages of changing designs and implementations. In using the object language model there are two outstanding issues: whether dynamic or static-typed object-oriented languages are suited to building applications, and the use of metadata to describe building objects.

2.4.1 Static or Dynamic Types

The prominent commercially available OO languages are C++ and SmallTalk (Smalltalk, 1990). Amongst the programming community C++ has become popular. The STEP Express language is a static-typed data exchange specification language. Many object-oriented languages have been developed within research organisations. Within New Zealand, a dynamic classification language called Kea has been developed to computerise building codes (Hosking et al, 1991).

A workshop on object-oriented languages (Palsberg and Schwartzbach, 1991) concluded that there is much commonality amongst languages, but the major distinction was whether the language had dynamic or static typing. The term "type" has broad meanings. In its widest sense it is a general categorising statement that an object fulfils certain requirements. In the narrower computer modelling sense it refers to a list of attributes and behaviours (functions). In a static typed language such as C++ an object is defined as belonging to one class (i.e., the class is the type), whereas in dynamic languages (e.g. SmallTalk) the object can be classified to a number of types as the program runs.

Dynamic typing is used where the object's state, role, or involvement (the object is placed in different environments) changes. The domain of building has many uses for dynamic typing. A typical building object state change example is the load-bearing and non load-bearing wall. A role change is the different attributes of a room depending upon its role. From a fire-code application (Hosking et al, 1987) an assembly room will have an occupancy variable named number_of_chairs whereas a dormitory's occupancy would be number_of_beds.

The dynamic-typed language is used where an object is assigned different attributes and different behaviours during that object's existence. Static-typed languages do not have this power of expression, but are still applied to BPM for their efficient implementation. A number of OO analysis and design authors avoid dynamic typing by the use of delegation (e.g. Rambaugh, 1991). With delegation, different states, roles and involvements are encoded as enclosed classes to which are the implementation of new behaviour is delegated, rather than by creating new types (classes).

Where the requirements for dynamic typing can be avoided, static-typed language implementations of BPMs will play a major role in efficient BPM engines. Dynamic typing can be simulated in a non-object-oriented way by having a type variable that is used to cause different behaviour.

2.4.2 MetaData

Although it has been stated that geometry alone can not be the basis of BPMs (Eastman, 1980), the most detailed computerised descriptions of buildings exist in CADD databases as unstructured collections of graphic entities. What is meant here by detailed descriptions is the degree of complexity of the collection of objects as seen by a human viewing a CADD image. It is not necessarily the sophistication of the basic objects (i.e. lists of attributes). The degree of structure varies, for example named layer structures (AutoDesk, 1994) organise collections of basic 2D graphics elements to represent buildings. Within the volumes of building literature, building elements are seen to be more rearrangements of basic objects rather than unique objects in themselves.

Figure 2 show types of building objects that are arrangements of components. These suggest that the classes to define these arrangements are best modelled as metadata classes rather than distinct classes. A metadata class is defined here to be a class derived from a base class, without any extra structure. Instances of a metadata class have specific values defined. As well, a metaclass can modify the base classes' behaviour. The wall metadata class example enables specific types of walls to be derived from the generic wall. The specific wall may not have any new attributes, rather the generic wall sytem components, dimensions and size attributes are instantiated, and certain behaviours set.

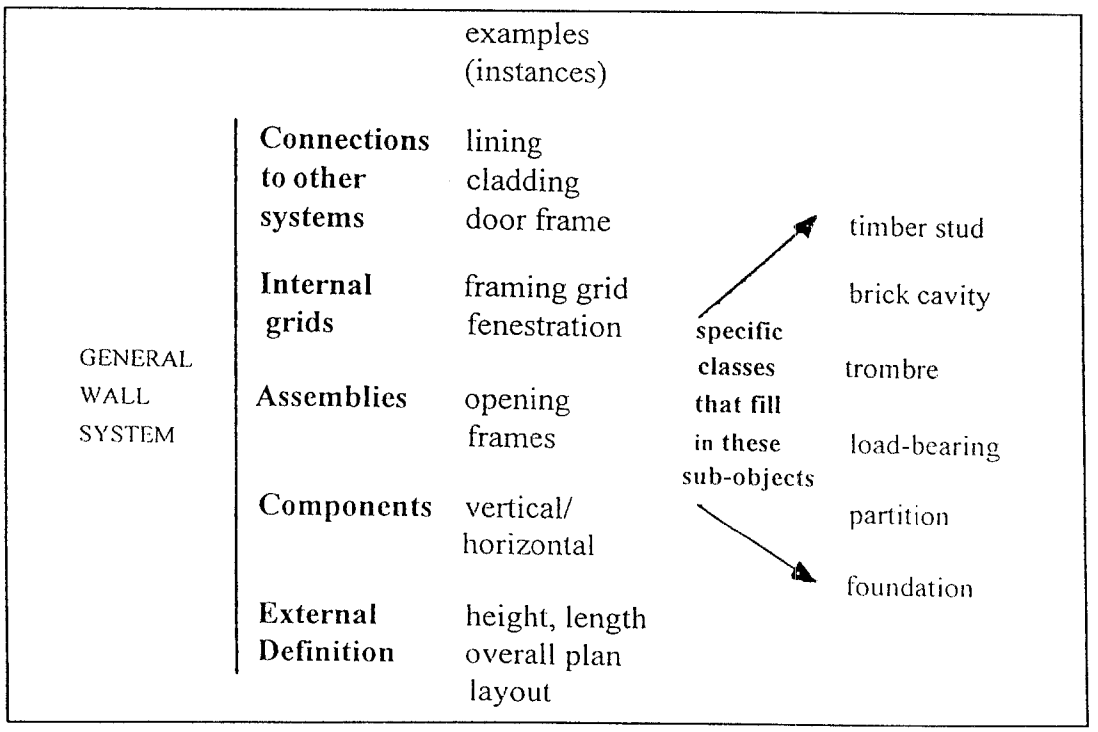

Figure 2: Some BPM MetaData Types

3. SELECTED OBJECT-ORIENTED DATABASE FEATURES

Having mentioned the development environment that encompasses the analysis, design and implementation of BPM models, language features and inter-schema mappings, what remains are the basic low level details of the OO database mechanism. The higher level CASE features of OO databases are set aside here because there are important low-level issues relating to implementing BPM class library databases. It is at this level that current OO database products need to be compared. Specific references to particular OO databases are not quoted, as the author's work did not a include a full technology comparison, which requires many databases to be tested.

With regard to the impact of OO database technology, BPM class libraries are likely to have:

1. Complex inter-class relationships, multiple inheritance (a class derives from more than one class) etc.
2. References to objects that are not to be stored. They arise in the context of the computer operating system in which the object is placed, or the object makes a shared reference to another object.
3. Complex inter-linked groups of references to objects which entail what the author terms "pointer swarms". For example the double-linked list (a particular way of implementing lists) class will be a common implementation for a grid of points. The size of the groups of objects can easily grow beyond the size of the computer's memory. The OO database/operating system setup must manage large groups of referenced objects without attempting to pull all referenced objects into the computer's main memory at once.
4. The design of the class library includes multi-user concurrency design. How do we do versioning and the management of extended use of shared objects?

3.1 OBJECT ID , VERSIONING AND CONCURRENCY ISSUES

3.1.1 Object ID and Versioning

In an environment of distributed databases, objects must have correct identification. Pointers are commonly used to identify objects within a program. A pointer contains the location of the object in the computer's memory. Although pointers are hidden from the BPM class library user by the class interface mechanism, they are a necessary language feature for the BPM class library developer.

This notion of using pointers to indentify objects is quite limited; location is not identity. The introduction of object identification (OID) for an object corrects this shortcoming. The use of pointers requires a mechanism to translate pointers to objects in a computer's memory to the correct object in the database. The translation from a pointer (addressing a computer's memory) to a longer OID form (address a huge space of all databases, versions etc) is termed "swizzling".

Object identity (OID) and the OO database mechanisms related to it should provide several features necessary in a shared and distributed database.

The first is the provision of a unique identity of an object amongst all the copies of that object within any number of databases. When an object is altered in a computer, and past states are to be saved (versioning), or when it is moved or shared, the object is copied. Usually object identity refers to the copy of the actual object; it is a value instance. OID should refer to the actual object, not any particular copy. The distinction between the actual object and copies of the object within a computer is an extension of conventional object identity (Godard and Simmel, 1994).

An important OO database feature closely associated with object identity is how an object is retrieved from the database. A database with different forms of identification for the actual object and groups of copies requires greater flexibility in how an object is addressed. This flexibility includes object retrieval given only partial addressing information.

The exploration of the possible designs for the concurrent engineering BPM environment is required in further BPM research. An example of the diversity of design exploration required comes from a panel discussion (Kent, 1988) on the issues relating to object versioning. Kent concluded that the numbers of different approaches to versioning are as many as there are participants and different attributes that require versioning.

3.1.2 Concurrency

Concurrency is the issue of how to manage more than one access to the database at the same time. Transaction processing is a feature that is offered by the traditional databases that claim to correctly cater for traditional concurrency requirements. There are two requirements which transaction processing databases provide. First is the assurance that data transfer is either completed (committed) as a whole or not done at all (partial database manipulations are undone or rolled back). This ensures database consistency. Secondly, each user of the database sees only their actions, in that the state of the database is frozen so they can effectively carry out their actions. Transaction processing is implemented by juggling data locks, and correctly serialising access to the data.

Multi-user building design networks will require transaction processing schemes that depart from the traditional database. The assumption that transactions are short (hence data can be just locked) is extended to quite long "grabs" of the data. Due to the long transaction the commits will need to be nested. Perhaps a concurrent design network will require even more complex transaction schemes than these. Experimentation with new transaction schemes could be required. Most OO databases offer the long and nested transaction processing models as a fixed database feature.

3.2 IMPLEMENTING INTER-OBJECT RELATIONSHIPS

Pointers are used in class libraries to implement inter-object relationships. A very important database requirement is to maintain database consistency between linked objects. The database term is "referential integrity". Due to the complexity of objects this requirement becomes vital in OO databases. The common situation is a bi-directional relationship. One object has a reference to another object which in turns refers back to the original. In a database that maintains referential integrity, in one of the objects the inverse reference is flagged with the INVERSE keyword. The database then manages this bi-directional relationship. The mechanisms to do this involve global garbage collectors that delete unwanted objects, and generally follow object references.

Delegating the maintenance of these relationships to the database is one approach. The other approach is seen in the design of container classes in which relationships between the contained objects are managed as the major part of the container class design.

There are pros and cons for both approaches. Those who advocate the use of INVERSE reference flag state that specifically programming for complex relationships amongst a collection of objects is a burden. The use of constraint programming applied to database environments that contain multiple data schemas (Amor, 1994) illustrates other language features that are needed to maintain the high end of inter-object relationship complexity.

On the other hand, there are two main reasons for designing the managing of inter-object relationships within a container class. The first is that a global database does global scanning of all objects maintained by it. This implies that it is an expensive operation, requiring occasional action. Such part-time global reference maintenance may not be adequate for the needs of large OO database applications that have frequently invoked object tranversal algorithms. On-screen display of joint objects (objects that are created as the joint between two objects) is a prime example.

The other reason is that there are various ways of implementing inter-object references. One of the author's favourite mechanisms is the joint object that contains pointers to the connecting objects. Joint objects are a very useful design idea,. They are central to the general AEC reference model (GARM), (Geiling, 1990). In this scheme no pointers **to** the joint objects exist. Not having bi-directional references reduces class relationship complexity. Reference following will not work to remove deleted references in this situation. The garbage collector has to know to look into the joint objects themselves, which is not possible without complex global object relationship tables.

These two approaches will be combined in a sophisticated BPM class library.

The last aspect of inter-object relationships is the combination of objects that are to be stored in a database (persistent) and objects that are part of the operating system that cannot be stored (transient). The common example of an object for which it is pointless to attempt to store is the window in which the object appears on-screen. Windows are examples of objects that are transient; they are created as a program runs. The OO database must cater for this. The transient objects are labeled. The OO database doesn't attempt to store them and when loading the object into the computer memory, the association with the transient object must be linked.

3.3 THE INTEGRATION OF CLASS LIBRARY, THE OBJECT-ORIENTED DATABASE AND THE OPERATING SYSTEM

The remaining features to be discussed are the major differences between OO databases. They impact on the integration of the class library, OO database and the operating system. Central to the software component alternative to neutral data exchange is the portability of the class libraries between developers who use the library in different environments. To aid portability, the impact of adding persistence on source code must be kept to a minimum. Any degree of standardisation will be very important to BPM class library reuse. Is there any real standardisation occurring within OO database technology?

The major difference in current OO databases is how objects are made persistent (lasting longer than their existence within a program). The two approaches are:
1) Specific load/store operations occur within the code, and
2) Transparent (the OO database takes care of it).

In the second approach, the database is closely linked to the operating system. The technology of virtual memory in current operating systems is expanded upon. The address space (the number of objects able to be addressed) of the computer is expanded so that many different databases can be uniquely addressed. The effect is that specific load/store data transfer operations between the database and the computer memory are not required for an object. There are different designs as to how objects are clustered together when the system "pages out" memory to the database (Wilson and Kakkad, 1992). The choice is between storing the individual object or a page containing the object. These, however, only affect database performance (hence this is what is usually debated), and do not have a large impact on the coding of class libraries. What does make a difference is whether you have load/store operations or not.

The impact of differing OO databases upon class libraries continues with what is termed the data definition language. In the context of making a C++ class library persistent, the C++ language is extended. Such additions include special keywords, special persistent classes and extra SET objects that refer to persistent collections.

These choices do not indicate much standardisation at all. Perhaps winners will have to be chosen. The structure of BPM class libraries is very dependent upon

what technology is chosen. There are two major standards within object-oriented databases. They are the ODMG-93 (Object Data Management Group) (Strickland, 1993) and COBRA (Common Object Request Broker) (Corba, 1991). Neither attempt to address the afore-mentioned issues. The current version of the ODMG standard presents definitions for an object and how to handle referential transparency. COBRA defines a protocol for accessing networked objects.

4. FURTHER WORK

Currently, a BPM application is being written that enables grid-based building objects to be manipulated and stored. This will demonstrate the half of the basic OO database features mentioned in this paper. What remains is the demonstration of the utility of correct object identification in a concurrent engineering environment and the development of a detailed metadata class-based BPM.

The methodology involved in a multi-schema mapping language of Amor (1994) will be adapted to C++. This will enable internal schema mappings within a BPM engine to be consistent.

5. CONCLUSION

In the context of development and delivery of BPM via the path of class library creation, the author asserts:

- That static-typed class libraries will contribute to the development and delivery of BPMs.
- That implementation is important as well as defining the basic BPM structures (classes and class relationships). This is so even though implementation-independent descriptions are popular amongst BPM researchers.
- A class library can be created that has persistency features required for distributed and concurrent computing that will be central to BPM applications.
- OO database standardisation is unlikely given the diverse approaches existing today. How the different OO databases applications can share data is a major problem, considering the diversity of current OO database systems.

It is important that the central ideas within diverse OO database technologies be discussed in the context of how BPM should be developed and delivered. This will come as more OO database experimentation is carried out within BPM development.

6. REFERENCES

Amor, R. W. 1994. *A Mapping Language for Views*. Department of Computer Science, University of Auckland, Internal Report. 33pp.

Amor, R. W., Augenbroe, G., Hosking, J. G., Rombouts, W., Grundy, J. 1994. *Directions in Modelling Environments*. Submitted to Automation in Construction. ISSN 0926-5805.

Augenbroe, G. 1992. COMBINE: *A Joint European Project Towards Integrated Building Design Systems*. Proc. Building Systems Automation-Integration '92. pp 10-12, Dallas, Texas, USA.

AutoDesk. 1994. *Layer Naming Convention for CAD in the Construction Industry Version 2*. AutoDesk Ltd

Bjork, B. 1989. *Basic Structure of a Proposed Building Product Model.* Computer-aided Design. Vol. 21(2), pp71-78.
CADKEY, 1994. *CADKEY Object Development Package.*
CORBA, 1991. *The Common Object Request Broker: Architecture and Specification.* OMG TC Document Number 91.12.1 Dec 1991.
Dechapunya, A. H, Hosking, J. G., Mugridge, W. B. 1990. *From Firecode to ThermalDesign: KBS for the Building Industry.* New Zealand J. Computing. Vol2(1). pp23-32.
Email. 1993. *Various STEP critiques posted on the express-user email mailing list and others.*
Eastman, C. M. 1980. *Prototype Integrated Building Model.* Computer-aided Design. Vol. 20, No. 3, pp137-145.
Forcade, T. 1993. *Evaluating 3D on the High End,* Computer Graphics World, Vol. 16, No. 10 & 11. Oct/Nov 1993, PennWell Publishing Company, USA.
Gieling, W. 1988. *General AEC Reference Model.* ISO TC184/SC4/WG1, document number N329.
Godard, I., Simmel, S. 1994. *Kala Version 3 Manual.* Penobscott Development Corporation. Cambridge, MS, USA.
Hosking, J. G., Mugridge, W. B., Hamer, J. 1987. *FireCode: A Case Study in the Application of Expert System Techniques to a Design Code.* Environment Planning and Design B. Vol. 14 pp267-280.
Hosking, J. G., Hamer, J., Mugridge, W. B. 1991. *Kea 1.0 Tutorial Manual* BRANZ Contract 85-024 Technical Report No. 18. Department of Computer Science, University of Auckland.
Hosking, J. G., Mugridge, W. B., Hamer, J., Amor R. W. 1994. *An Architecture for Code of Practice Conformance Systems.* VTT Symposium 125, Computers and building Regulations. pp171-180.
Intergraph. 1989. *Software Catalog.* Spring 1989. Intergraph Corporation. Printed in the USA.
Karasick, M. 1989. *The Same-object Problem for Polyhedral Solids.* Computer Vision, Graphics, and Image Processing. Vol . 46. Academic Press, Inc. pp23-36.
Kent, W. 1991. *An Overview of the Versioning Problem.* SIGMOD proceedings.
Nijssen, G. M., Halpin, T. A. 1989. *Conceptual Schema and Relational Database Design: A Fact Oriented Approach.* Prentice Hall.
Palmer, M., Gilbert M., Anderson J. 1992. *Guidelines for the Development of STEP Application Protocols.* A STEP draft document.
Palsberg, J., Schwartzbach, M. 1991. *Three Discussions on Object-oriented Typing.* Report on ECOOP '91 Workshop W5. ECOOP proceedings.
Pixar Corporation. 1988. *The Renderman Interface. Version 3.0.* Ranfael, CA, USA. May 1988.
Price, C. G. 1993a. Personal E-mail Communications.
Price, C. G. 1993b. *A Building Data Model for Integration.* Study Report No. 49. BRANZ.
Price, C. G. 1994. *The Development of Class Libraries for Building Product Models.* Technical Computing, The ACADS Journal, No 80. March 1994 . pp13-15.
Requicha, A. A. G. 1980. *Representations for Rigid Solids: Theory, Methods, and Systems.*
Rumbaugh, J., Blaha, M., Premerlani, W, Eddy, F. Lorensen, W. 1991. *Object-oriented Modelling and Design.* Prentice Hall. ISBN 0-13-629841-9.
Silicon Graphics. 1992. *Open GL Reference Manual.* Silicon Graphics Inc. ISBN 0-201-632764. Addison-Wesley Publications.
SmallTalk, 1990. *SmallTalk Language Reference Manual.*
SQL. 1984. *American National Standard Database Language SQL: Working Draft.* Document X3H2-85-1. ANSI, New York. December 1984.
STEP, 1991. *Part 1: Overview and Fundamental Principles.* A STEP draft document N14. ISO TC184/SC4/WG4
Stroustrop, B. 1990. *The C++ Programming Language. Version 3.0.* Addison-Wesley.
Strickland H. 1993. ODMG-93: *The Object Database Standard for C++.* C++ Report. Oct 1993. SIGS Publications. USA. pp45-48.
Turner, J. A. 1989. *A Systems Approach to the Conceptual Modelling of Buildings.* Architecture and Planning Research Laboratory, University of Michigan, Ann Arbor, USA. 12pp.
Wilson, P. 1993a (Latest Version). *Processing Tools for Express.* Online Document available on the STEP Online Information System (SOLIS).
Wilson, P. (Editor). 1993b. *Information Modelling in Express.* Rensselaer Polytechnic Institute. Troy, NY, USA.
Paul Wilson, Kakkad S. 1992. *Pointer Swizzling at Page Fault Time: Efficiency and Compatibility Supporting Huge Address Spaces on Standard Hardware. An online document, Department of Computer Sciences, University of Texas, Austin, USA.*