JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No.2, March - April 2005

Association Implementation

The Key to Efficient Persistence

David Johnson, Lecturer, West Coast College TAFE, W.A., Australia

Abstract

The Object-Relational (O-R) approach to persistence is alive and well. It is based upon the proven and reliable relational database approach, and provides choice from a range of well-supported Relational Database Management System (RDBMS) packages. Major compiler and systems support players continue to offer increased features (e.g. object data-type definition and storage support in Microsoft's ADO.Net in Windows 2005), which are supportive of the O-R approach and contribute to a healthy O-R future.

With the appropriate design techniques and persistence software support, the O-R approach represents the fastest, most reliable way to develop efficient, scalable, maintainable applications software. An added bonus is that the skill set of developers transferring from Relational-Procedural to Object Oriented (OO) application development is well suited to the O-R approach.

With appropriate persistence support software, business object mapping and persistence can be automated, freeing application developers from the time-consuming tasks of mapping objects to database tables and the (re)writing of database access code. The O-R approach can be a SQL-free experience, allowing developers to concentrate on CRUD (Create, Read, Update and Delete) access and processing of individual or groups of business objects without having to worry about the underpinning relational database framework being used to persist business object data.

The object-oriented application development becomes transformed into a simple 3-step process:

1. Develop an Implementation Object Class Model,
2. Generate the Business Objects and Object Database for object persistence, and
3. Code and test the application based upon the Business Objects.

The key to success for this development process lies in the first step, 'develop an Implementation Object Class Model', because the way in which object class associations are implemented can significantly affect the performance of applications that depend upon them. Conversely, any time and effort spent producing a good business class design can be completely negated in terms of object access performance due to poor association implementation.

Cite this article as follows: David Johnson: "Association Implementation", in *Journal of Object Technology*, vol. 4, no. 1, March - April 2005, pp. 81-107.
http://www.jot.fm/issues/issue_2005_03/article1

The UML design process, which is pivotal to any OO development, places considerable emphasis upon the identification and differentiation of associations between objects. One significant weakness of the UML design process is the provision of too little support for implementation planning too late in the development process.

This article tackles this problem by introducing some simple high-level implementation semantics for association implementation using the UML Object Class diagram. It discusses data structures and logic support to underpin various alternatives using an O-R approach, and shows how implementation planning for associations can fit into the application development cycle to form the basis of an agile, practical and targeted OO application development methodology.

1 INTRODUCTION

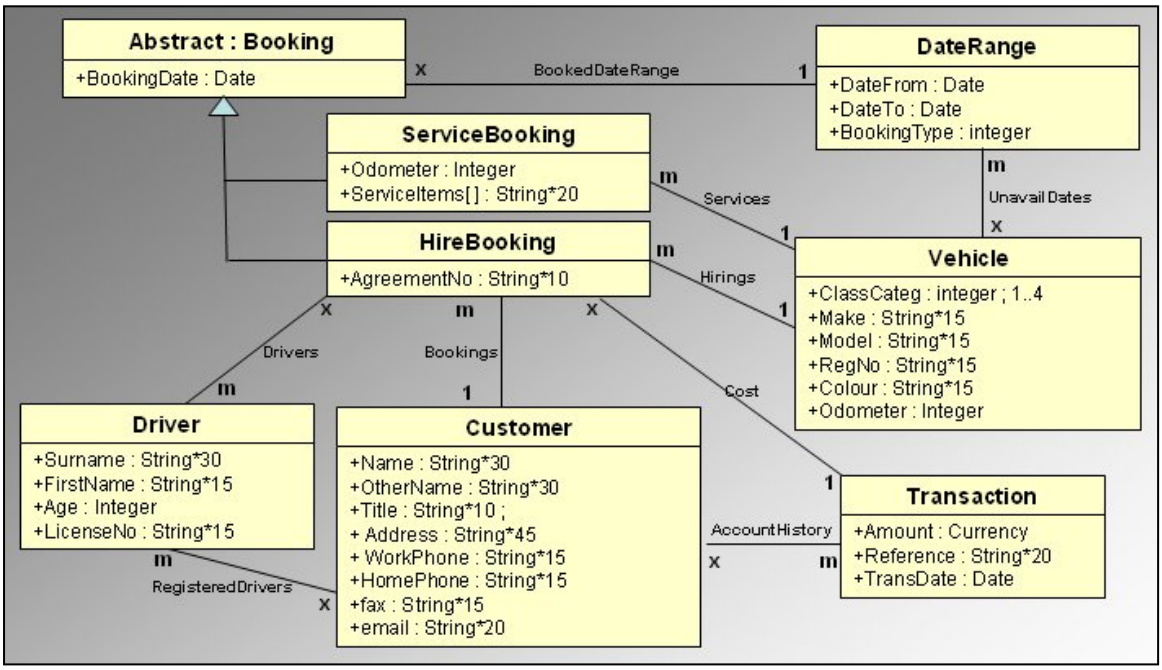
UML Object Class diagrams represent object classes together with their public properties, private and protected data structures, method headers, inheritance hierarchies and associations. Implementation of the object class code from these diagrams involves the coding of properties, private data structures, methods, and inheritance hierarchies, together with business rule logic. These processes are well addressed in OO texts. The ways to provide for persistence and to implement associations are less well documented.

Persistence relates to the way that object data is stored and retrieved. We shall be exclusively considering object persistence using an O-R approach, with the data for each leaf object class in the inheritance tree being stored in its own table. This approach is sometimes referred to as horizontal partitioning. Other schemes are possible and are well covered in the existing literature, but horizontal partitioning is simple to implement, does not require cascading database calls to support inheritance, and, in combination with object caching, results in fast object data access times and reduced network traffic.

To increase consistency between objects and applications it is advisable to provide a unique ObjectId (Ambler's OID) property for each object class, even when there is a naturally occurring one (eg. CustomerId, AccountCode). The ObjectId, which should be unique across object classes, is used to identify an object's data in the object database.

A UML Object Class diagram, such as that below for the Car Rental/Servicing application, represents data structures, associations and inheritance between object classes.

Object Class Diagram for Hurtz Car Rental Application



Associations indicate that an object of one class needs access to other objects. For instance, a **HireBooking** object needs access to the associated vehicle, a transaction, customer, a **DateRange** (via inheritance), and possibly several driver objects. The multiplicity (or ordinality) in each direction needs to be clearly shown for both ends of an association. An 'x' should be used to indicate when one object does not need to know about the other (eg. A **HireBooking** needs to have access to all registered drivers, but each driver object does not need to know which **HireBooking**s reference them). Business Rules should be provided (possibly as attached notes) to justify the multiplicity shown and/or any validation checks that need to be made.

A UML Object Class Diagram clearly defines data structures (i.e. public, private and protected data members) and specific functionality (methods and internal functions) internal to each object class, and external factors such as inheritance and associations between object classes. The implementation of associations between object classes will require additional data structures and/or functions. No object class can be coded and used within an application until the way in which each association is to be implemented has been determined.

As well as requiring specific data structures and/or functions to implement associations, the implementation choice can have a major impact upon the processing overheads and data access speed and efficiency. The main questions that need to be addressed when determining the most appropriate way to implement each association relate to :

- Object retrieval: Is it Cached or Non-cached?
- Data return mechanism: Via Properties or Methods?
- Data type returned: As Objects or ObjectIds?
- Mapping technique: Use Foreign-Key or Index?

The key issues relating to these questions are addressed in the Section 2, and summarized in Section 3 to provide practical guidelines to simplify the implementation process.

2 ASSOCIATION IMPLEMENTATION FACTORS

Object Retrieval: Cached or Non-cached

Cached retrieval allows the forward loading and storage of object data. The advantages are reduced database access frequency (larger packages of data) with a consequential reduction of network traffic, database server loads and marshalling overheads.

Caching increases probability that object data will become out of date in a multi-user environment. This problem can be managed by the implementation of an appropriate time stamp and time-locking regime.

As well as providing timed-locking of objects, some commercial O-R support packages allow the access depth, and thus the depth of associated data, to be varied dynamically with application code. Thus, when an application provides user browse and select options, only the top level objects need to be loaded into cache (i.e. without any dependent objects), and when more information is required about a specific objects, the lower levels can be loaded, populating cache with deeper pyramids of object data.

Cached retrieval should be always be used on networked applications, and for the rest of this article, write-through cached retrieval will be assumed to all object retrieval.

Data Return: Property or Method

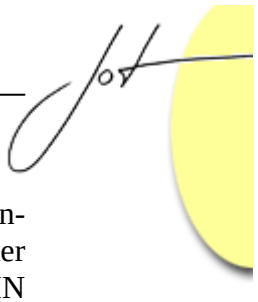
Associations may be implemented as properties or as methods. However, because objects are referenced via instance variables, it seems more appropriate to represent associations as public variables (i.e. as properties).

Weak, less frequently used associations can be implemented as methods, but this still requires provision of data structures (usually private) to access and/or store associated object(s), as well as a method to access associated data and methods.

The **method** approach thus offers no savings in terms of data storage or access efficiencies over the **property** approach, which is the recommended approach. The rest of this article assumes a property approach to association implementation.

Returned Data Type: Objects or ObjectIds

Object data retrieval and instantiation overheads can be minimised by only loading associated object data when frequent navigation of an association is required, and by



retrieving only the ObjectIds when infrequent navigation applies. Thus an application-specific FIN Analysis, classifying expected association navigation frequency as either Frequent, Infrequent or Never, is an important association implementation task. A FIN Analysis table for 9 associations identified in the Hurtz Car Rental Object Class design (provided earlier) is provided below.

Association Name	Association Object A	Association Object B	Access Frequency	
			A to B	B to A
Bookings	Customer	HireBooking	Frequent	Frequent
RegisteredDrivers	Customer	Driver	Infrequent	Never
AccountHistory	Customer	Transaction	Infrequent	Never
Drivers	HireBooking	Driver	Infrequent	Never
Cost	HireBooking	Transaction	Infrequent	Never
BookedDateRange	Booking	DateRange	Frequent	Never
Hirings	HireBooking	Vehicle	Frequent	Infrequent
Services	ServiceBooking	Vehicle	Frequent	Infrequent
UnavailDates	Vehicle	DateRange	Infrequent	Never

It is recommended a **Frequent** classification be assumed if in doubt. This reduces the amount of special instantiation code that needs to be provided. Also, with the appropriate O-R software support, the frequency classification can be changed relatively simply before an application reaches the Release stage.

When only the ObjectIds of the dependent (i.e. associated) objects are loaded, as for infrequently navigated associations, the application developer need to provide data retrieval and object instantiation code if and when the association needs to be used.

When an association is implemented via an object property (multiplicity of **1**) or as an object-group property (multiplicity of **m**), the object property instantiation should be placed within the property's Get logic. As well as deferring instantiation until the object is really needed, this encapsulation technique means that no additional logic needs to be provided by the application to navigate the association.

Mapping Technique: Foreign-Key or Index

Foreign-Key mapping is the most commonly implemented association mapping technique because it can be directly implemented using standard relational database design and implementation techniques.

Associations with 1:m multiplicity are simply implemented as 1:m relationships between the object class data tables, with a copy of the independent object's primary key (on the **1**-side) is stored as a foreign key property in the dependent object (on the **m**-side). The **many** side is thus retrieved by an SQL Select operation selecting all dependent records with a foreign key corresponding to the independent's primary key.

To maintain the integrity of the Object database, **referential integrity** needs to be enforced on all Foreign-Key mapped associations. Also, because a change of foreign-key reference on the **1**-side is the only way the **m**-side can be changed, that the **m**-side of the relationship must be a read-only property.

In relational databases **m:m** relationships need to be converted into two 1:m relationships via an associative entity. Similarly, **m:m** associations implemented using Foreign-Key mapping require the creation of an associative object class.

Index mapping requires server-side and user-side logic to maintain separate indexes for associated objects within the database. Many indexing schemes are available (inverted indexes, balanced B-tree, ISAM etc.), but keeping in mind that index mapping is only part of the association mapping picture, the simpler the scheme the better.

Advantages of Index mapping are that associative object classes are not needed to implement **m:m** associations, that objects can be directly added or removed from the association (i.e. they are not read-only as with Foreign-Key mapping), and the ability to support multiple associations without a proliferation of unwanted foreign keys

The author is only aware of one commercial package, Design SCOPE's ObServer, which seamlessly supports both Foreign-Key and Indexed mapping of associations, and allows the depth of associated data to be varied dynamically during application execution.

3 ASSOCIATION IMPLEMENTATION GUIDELINES

A summary of the recommendations for association implementation discussed so far are:

- Use cached retrieval.
- Represent associations as properties.
- Use access-frequency estimates to determine whether ObjectId or object properties will be used for association implementation, as summarised below:

Association Multiplicity	High Access Frequency	Low Access Frequency
1	Object Property	ObjectId Property
m	Group of Objects Property	Group of ObjectIds Property

- When an association is implemented as an object property or as an object group property, object instantiation should be placed within the property's **Get** logic.
- Use Foreign-Key mapping for a 1:m associations.
- Use index mapping for m:x associations and for m:m associations where an associative object is not needed to provide extra data and/or functionality.

A notation to assist the transition from UML Object Class diagram to Implementation diagram, and some techniques to detail the data structures required to underpin the planned implementation will now be discussed.

4 UML ASSOCIATION IMPLEMENTATION NOTATION

The recommended UML Association Implementation notation involves:

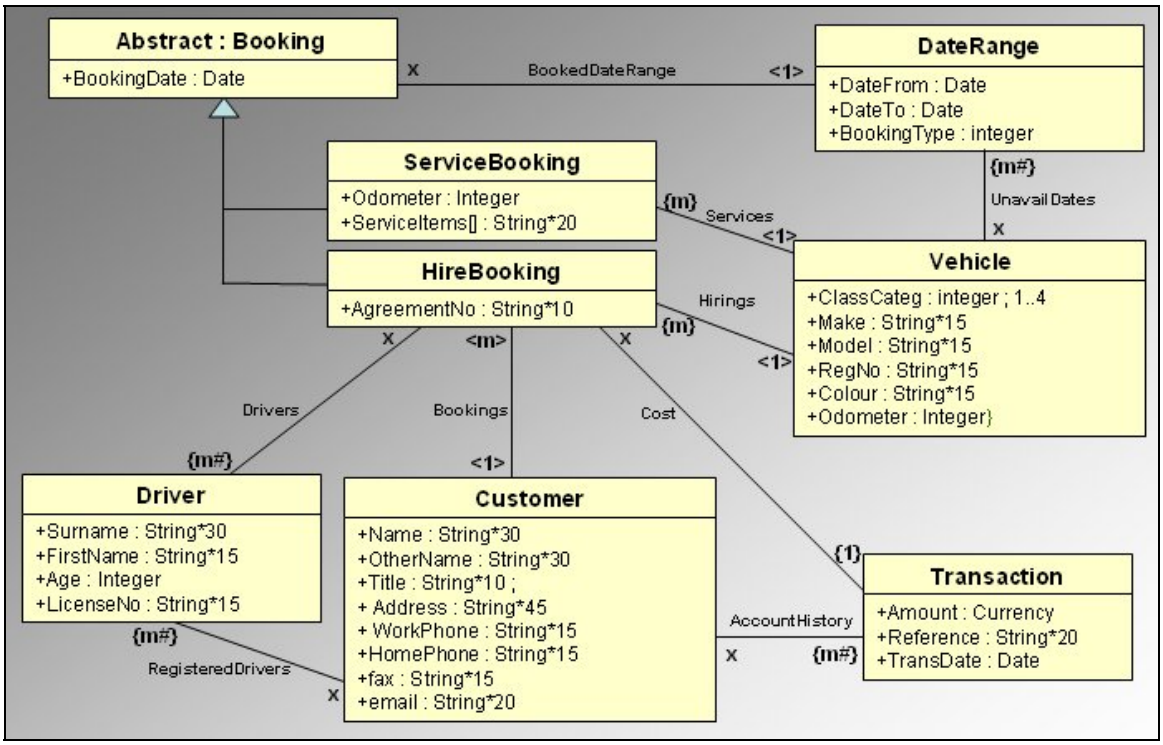
Angled brackets	**< >**	for object properties
curly brackets	**{ }**	for ObjectId properties
round brackets	**( )**	for object return via a method call
Cross hatch	**#**	for index mapping

This simple extension to the UML Object Class multiplicity notation concisely and unambiguously covers all possible implementation patterns, as detailed below :

Data Return	Data Type Returned	Mapping Technique	
		Foreign-Key	***Index***
String or numeric **property**	**ObjectId** of one object	{1} or 1{ }	-
Business object **property**	**Object**	<1> or 1<>	-
Group object **property**	**Group** of **ObjectIds**	{m} or m{ }	{m#} or m{#}
Group object **property**	**Group** of **objects**	<m> or m<>	<m#> or m<#>
Function **method**	**Object**	(1) or 1()	-
Function **method**	**Group** of **objects**	(m) or m()	-

UML Association Implementation Notation Summary

As both UML Object Class diagrams and Implementation diagrams are an important part of the design artefacts, the Object Class diagram should be copied and then be extended using the notation to create an intermediate Implementation Diagram (i.e. a work-in-progress version), as shown below for the Hurtz Car Rental UML example using the access frequencies listed earlier.

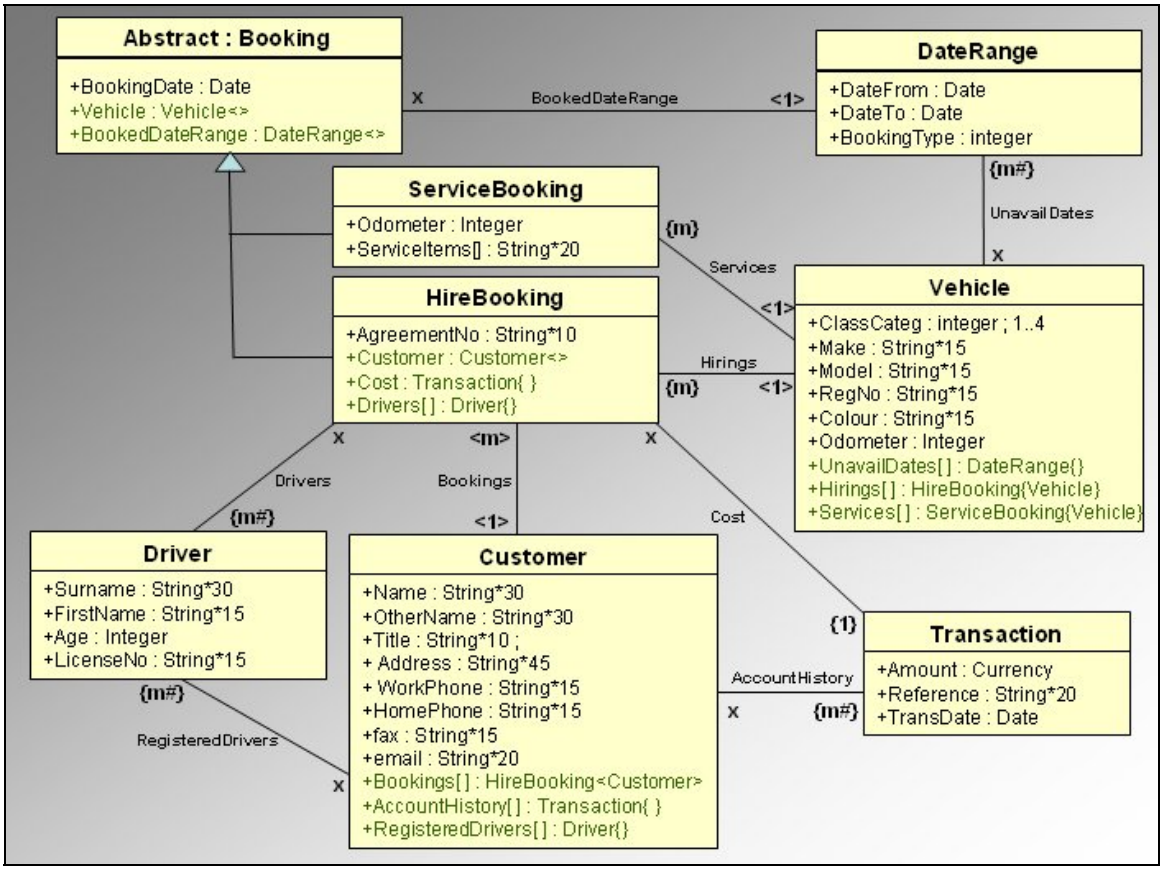


Intermediate Implementation Diagram for Hurtz Car Rental Application

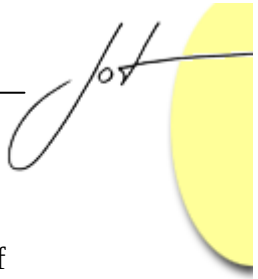
The implications of this design are that, for an access depth of 1, object data for all associated **HireBooking** objects will be loaded into cache automatically as a **Customer** object is loaded but only the ObjectIds of associated **Transaction** and **Driver** objects will be available. Separate logic is required to retrieve data and to instantiate the associated **Transaction** and **Driver** objects. However, with an access depth of 2, **Vehicle** and **DateRange** object data for the associated **HireBooking** objects would be automatically loaded into cache as well.

This notation is intended to allow various implementation scenarios to be quickly and easily explored in conjunction with evolving screen prototypes and Use Case diagrams and definitions. As the design starts to stabilise, the Implementation design can be completed by adding property data structures to the Object Classes which are needed to implement the association mappings.

For data structure definitions, square brackets [ ] are used to indicate properties of groups of ObjectIds or objects. When Foreign-Key mapping is used the foreign key property in the corresponding dependent (child) object needs to be identified as for the Hurtz example below.



Implementation Diagram for Hurtz Car Rental Application



Examining some of the property definitions in this examples in more detail,

+Customer: Customer<>	is an object property that is an instance of the Object Class Customer
+Cost: Transaction{ }	is an ObjectId property of Object Class Transaction
+Bookings[ ] : HireBooking<Customer>	is a group of HireBooking objects Foreign-Key mapped via the Customer property
+AccountHistory[ ] : Transaction{ }	is a group of index mapped Transaction ObjectIds

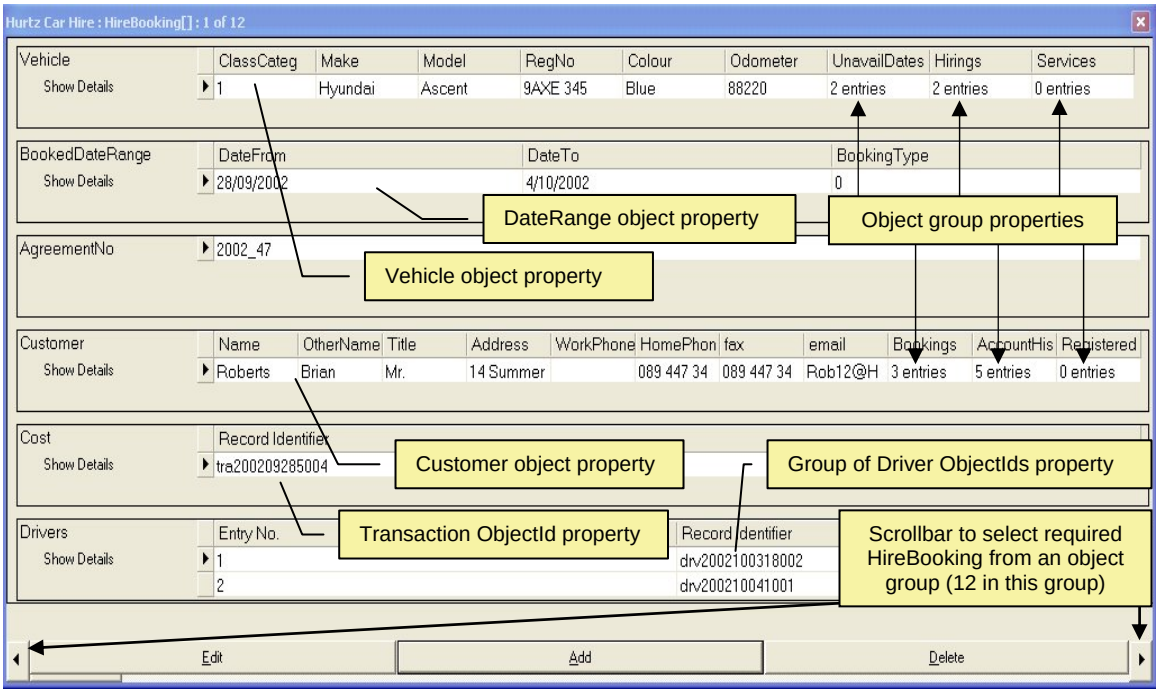
Notice that association names tend to appear as property names on the independent side of an association. Also it is good practice to, where possible, use singular names for object class and property names, reserving plurals for associative objects and for properties that represent groups of ObjectIds, objects or standard data types.

Once the supporting data structures have been added and cross-checked against the mapping notation, the Implementation diagram is ready for coding and testing.

Below is a VB.Net code extract generated for **HireBooking** by Design SCOPE ObServer.

```
Public Class HireBooking
    Inherits Booking

' Summary of all Properties...

Object Private/Protected Variables

#Region "Object Property Get/Set code"
    Public Property AgreementNo() As String
        ' > Property No. 3  ..... Design Definition = +AgreementNo : String*10
        Get
            Return gObServerPersist.GetProperty(3, Me)
        End Get
        Set(ByVal Value As String)
            ValidateProperty(3, Value, Me)
        End Set
    End Property ' AgreementNo

    Public Property Customer() As Customer
        ' > Property No. 4 ..... Design Definition = +Customer : Customer<>
        Get
            If objProp4 Is Nothing Then objProp4 = gObServerPersist.GetProperty(4, Me)
            Return objProp4
        End Get
        Set(ByVal Value As Customer)
            If ValidateProperty(4, Value, Me) = vbNullString Then objProp4 = Value
        End Set
    End Property ' Customer

    Public Property Cost() As String
        ' > Property No. 5  ..... Design Definition = +Cost : Transaction{ }
        Get
            Return gObServerPersist.GetProperty(5, Me)
        End Get
        Set(ByVal Value As String)
            ValidateProperty(5, Value, Me)
        End Set
    End Property ' Cost

    Public Property Drivers() As ObjectGroup
        ' > Property No. 6  ..... Design Definition = +Drivers[ ] : Driver{}
        Get
            Return gObServerPersist.GetProperty(6, Me)
        End Get
        Set(ByVal Value As ObjectGroup)
            ValidateProperty(6, Value, Me)
        End Set
    End Property ' Drivers
#End Region

Object Property Validation Code

Object Method code

End Class
```

and an auto-generated CRUD form for edit/browsing **HireBooking** objects.

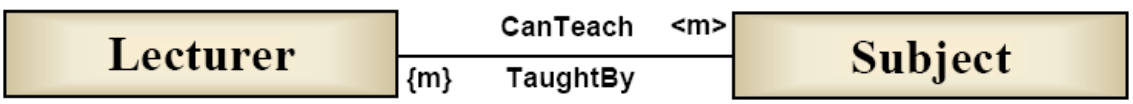


5 IMPLEMENTATION OF MANY:MANY ASSOCIATIONS

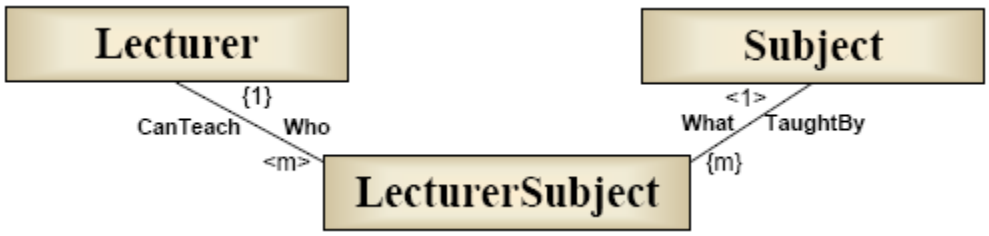
When using Foreign-Key mapping, **many:many** associations must be resolved by the use of Associative Objects and two 1:m associations in a similar way that associative entities are used in Relational Database design. For example, consider the association between **Lecturer** and **Subject** object classes wherein :

- a **Lecturer** object needs to know which **Subject**s the **Lecturer** can teach, and
- the **Subject** object needs to know which **Lecturers** can teach it.

The m: m association would initially be represented as:



To implement this m:m association using Foreign-Key mapping, an associative object, **LecturerSubject**, must be added, as shown below.

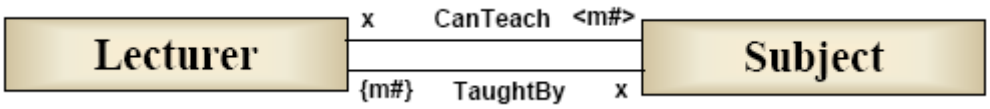


The resulting object class structure and processing examples are shown below:

Object Class Properties	Processing Examples
Lecturer<> Id : String Name: String CanTeach[ ] : LecturerSubject<Who> Matching foreign key properties **Subject<> Id : String Name : String TaughtBy[ ] : LecturerSubject{What}** **LecturerSubject<> Id : String Who : Lecturer{} What : Subject<>**	***Code to add the names of all subjects that a particular lecturer (myLecturer) can teach to a list box lstCanTeach.*** Dim LectSubject as LecturerSubject For each LectSubject in myLecturer.CanTeach LstCanTeach.Items.Add LectSubject.What.Name Next ***Code to add the names of all lecturers who can teach a particular subject (MySubject) to a list box lstTaughtBy.*** Dim LectSubjId as String For each LectSubjId in mySubject.TaughtBy Dim LectSubject = new LecturerSubject(LectSubjId) Dim Lecturer = new Lecturer(LectSubject.Who) LstTaughtBy.Items.Add Lecturer.Name next
Coding many:many Foreign Key Mapped Associations	

As with associative entities in Relational Database design, it may be appropriate and useful to attach other properties (eg LastTaughtDate and NumberOfTimesTaught) to the associative objects such as LectureSubject.

Index mapped many:many associations, however, can be directly represented without the use of an Associative Object as shown below.



with the corresponding object class property structure and usage example shown below:

Object Class Properties	Processing Examples
Lecturer<> Id : String Name : String CanTeach[ ] : Subject<> Subject<> Id : String Name : String TaughtBy[ ] : Lecturer{}	***Code to add the names of all subjects that a particular lecturer (myLecturer) can teach to a list box lstCanTeach.*** Dim Subject as Subject For each Subject in myLecturer.CanTeach LstCanTeach.Items.Add Subject.Name Next ***Code to add the names of all lecturers who can teach a particular subject (MySubject) to a list box lstTaughtBy.*** Dim LecturerId as String For each LecturerId in mySubject.TaughtBy Dim Lecturer = new Lecturer(LecturerId) LstTaughtBy.Items.Add Lecturer.Name next

Coding many:many Index Mapped Associations

Apart from the savings in terms of storage and maintenance overheads of an extra associative object, the above example demonstrates the savings in code simplicity when compared with the corresponding code for the associative object. Also, the TaughtBy[ ] property of the Subject object can be directly changed and replaced, whereas for Foreign-Key mapping this property's contents requires the changing of foreign key entries within purpose-specific associative LectureSubject objects.

6 THE COURSE INFORMATION EXAMPLE

The Hurtz example has been used to introduce the concepts, the notation and to demonstrate its use. However, design is not an exact science, and one textbook-styled example cannot fully address the design problems to be encountered in practice.

For instance, it can be difficult to recognise a close-to-optimal solution, let alone create one. Similarly, feasible solutions developed using theoretically correct interpretations and assumptions may result in unduly complex and inefficient implementations.

The Course Information application was developed to maintain Course Information for my college's IT handbook. It will be used to provide some practical advice and guidelines that may prove useful to help overcome problems that will arise with real world applications. Below is an extract from the handbook for the combined C117/C119 course.

C117 Certificate III in Information Technology (Software Applications)
C119 Certificate III in Information Technology (Network Administration)

Curriculum hours: 540 One semester full time or equivalent part time

COURSE SCHEDULE

Skill Set Names	SIN	TPN	Training Package (TP) Unit of Competency (UOC) Names	Core or Elective	Hrs
User and Technical Documentation	C1080	ICAITD128A	Create user and technical documentation	CoreBoth	20
Provide Advice to Clients	C1050	ICAITS031B	Provide advice to clients	CoreBoth	30
Office Advanced	C4535	ICAITU126B	Use advanced features of computer applications	CoreBoth	40
(Word Advanced, Excel Advanced, Access Advanced)	C4532	ICAITU018C	Develop macros & templates for clients using standard products	C117Core C119Elect	40
	C4533	ICAITU019C	Migrate to new technology	C117Core C119Elect	30
Customising Software	C4534	ICAITU028C	Customise packaged software applications for clients	C117Core C119Elect	40
Introduction to Web Programming	C1029	ICPMM65DA	Create Web Pages with Multimedia	ElectBoth	50
	C4470	ICAITB137A	Produce basic client side scripts	ElectBoth	30
Intro to Programming VB	C4570	ICAITB166A	Create Utility Programs	ElectBoth	30
Introduction to Networking	C4500	ICAITS120C	Connect Internal hardware Components	C117Elect C119Elect	30
	C1074	ICAITS121A	Administer network peripherals	C117Elect C119Core	20
	C4499	ICAITS020C	Install and optimise system software	CoreBoth	20
	C4495	ICAITI101B	Install and manage network protocols	C117 Elect	20
	C1047	ICAITS025B	Run standard diagnostic tests	C119 Core C117 Core	20
	C1053	ICAITS034B	Determine and Action Network Problems	C119Core	40

Office Advanced – 70 hrs

C4535
Manipulate data
Access and use support resources
Configure the computing environment

C4532
Determine macro and template requirement
Develop macro or template for client
Provide client support for macro or template

C4533
Apply existing knowledge and techniques to new technology
Apply advanced functions of the technology to solve organisational problems
Apply new functions of upgraded technology

Provide Advice to Clients– 30 hrs

C1050
Analyse client support issues
Provide advice on software, hardware and network
Obtain client feedback

Introduction to Web Programming – 40 Hours

C1029
Identify the Tools and Parameters of Web Page Design
Produce Web Pages
Validate and Prepare for Distribution

C4470
Construct a Script using Basic Syntax
Write Scripts using Methods, Functions and Events
Test Scripts and Debug
Create Objects for Dynamic Web Pages

Course Information Booklet Extract

The Course Information Example: Developing the Logical Object Class Model

The purpose of the Logical Object Class design is to develop a compact, unambiguous model that provides objects with access to all the data and with all the functionality they need to service the target application(s).

All properties, methods and associations identified for an object class should be in response to, and part of the answer to, the one basic question :

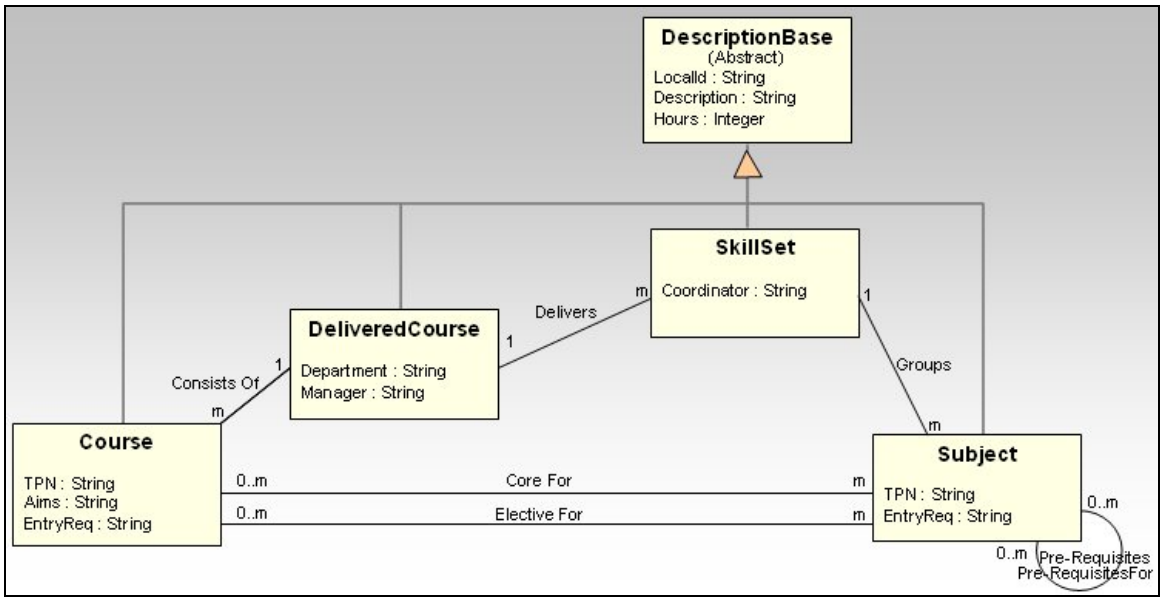
"What does an object of this class need to know and be able to do?"

This question should be asked time and time again in the object class design process.

What an ***object needs to know*** relates to its internal and exposed (property) data structures, and what it ***needs to be able to do*** relates to its internal and exposed (method) code-specific behaviour. Object data (inclusive of associated objects) and object functionality are the essential aspects of object class design.

The early stage of Object Class design involves the development of the **Domain Model**. Although ultimately important, functionality is only a minor concern for Domain Modelling. It is quite easy to add/modify method logic at any stage. Similarly, consideration of how the data is to be stored and retrieved from a database is of little concern for the Domain Model, and should be left until the implementation design.

For the Course Information application, some properties were identified as common to the **Course**, **DeliveredCourse**, **Skillset** and **Subject** objects, and were thus placed in an abstract Object Class named **DescriptionBase**, as in the Domain Model below.



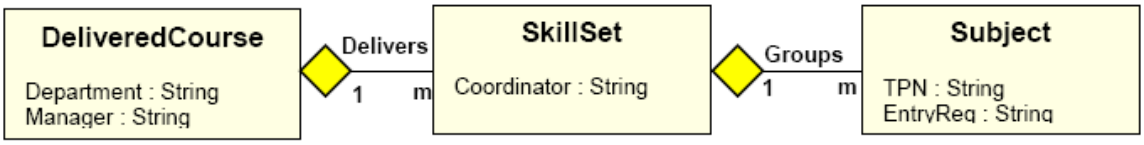
Logical Object Class Model (Domain Model)

Unless significant processing or flexibility advantages can be identified, there can be many feasible design alternatives. However, some of the tell-tale signs that a feasible design is a poor choice or is incomplete, and should thus not be implemented are:

- The design does not provide a good, comfortable fit to the data.
- The design is overly complex, with some team members having difficulty coming to grips with the design.
- Difficulty in reconciling particular application objects with Object Classes.
- Properties exist that do not fit comfortably into any specific Object Class.

The process of bouncing ideas between the Object Class design, the Use Case Model and the prototype until they all agree and are fully supportive of each other is strongly recommended. Only when this process stabilises, and the design becomes much simpler, more comprehensive and more robust that the commencement of the implementation planning can be contemplated.

Some may, quite rightfully, argue that some associations represent and should thus be shown as **aggregation** in the final domain design, as below.



Although correct in theory, in practice such detail will have no bearing upon either the implementation or object usage. The author thus feels that the time can be better spent on other aspects of the design, such as the design implementation. However, should you believe the careful delineation between types of association to be useful, then certainly no harm will result from adding it to your Object Class design.

The Course Information Example: Developing the Implementation Model

Below is a FIN (Frequent, Infrequent, Never) Analysis table for the 7 associations identified in the Object Class design (i.e. the Domain Model).

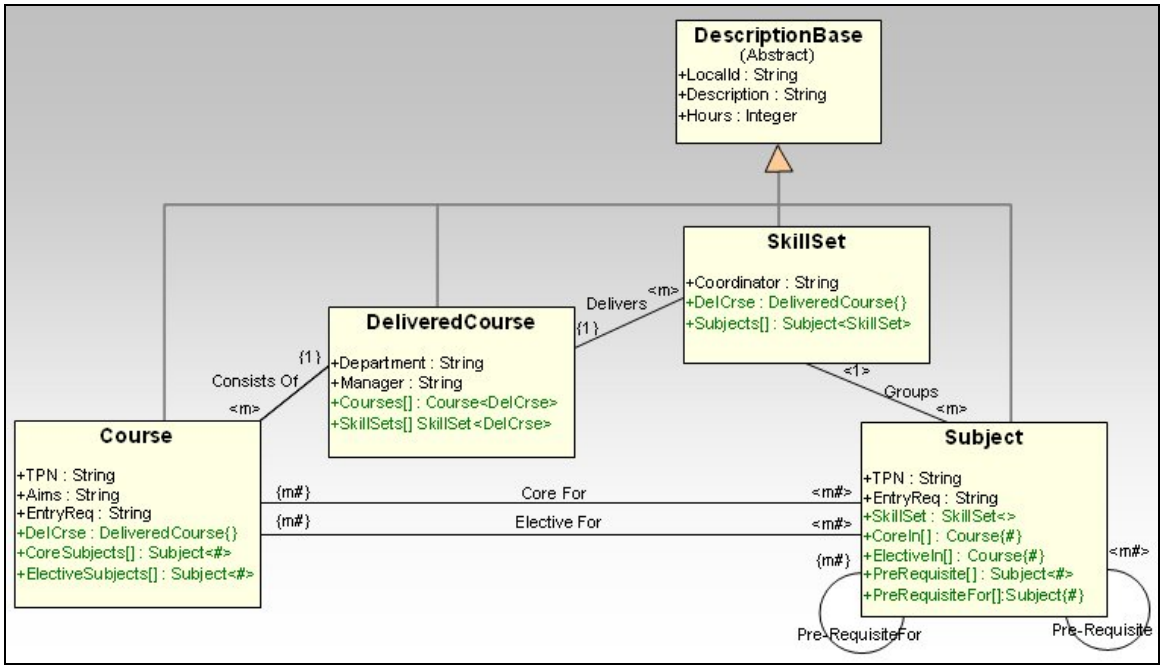
Association Name	Association Object A	Association Object B	Access Frequency	
			A to B	B to A
ConsistsOf	Course	DeliveredCourse	Infrequent	Frequent
Delivers	DeliveredCourse	SkillSet	Frequent	Infrequent
Groups	SkillSet	Subject	Frequent	Frequent
CoreFor	Course	Subject	Frequent	Infrequent
ElectivesFor	Course	Subject	Frequent	Infrequent
PreRequisites	Subject	Subject	Frequent	Never
PreRequisitesFor	Subject	Subject	Infrequent	Never

While a FIN table provides a neat summary of access frequency for documentation purposes, in practice it is usually easier to work on the evolving Implementation diagram (initially just a copy of the Object Class diagram), showing the relative navigation

frequencies by the < >, **{ }** and **x** notations. This notation is quick and easy to add and change, and provides a more understandable visual format for frequency allocation.

An important source of information about access paths and probable frequencies is the pseudocode or Action diagram logic from the Use Case definitions and/or a close examination of the processing requirements and patterns of the prototype screens. If in doubt, err towards an assumption of frequent access because this minimises any extra code required to instantiate an object from an ObjectId, but it may mean that object data might be automatically loaded into cache when you don't need it.

Implementation Modelling involves the two stages. The first is the addition of the navigation frequencies to the Object Class diagram to create an Intermediate Implementation diagram, and the creation and checking of a FIN table. The second is the addition of the data structures supporting the association mappings to the object classes to complete the Implementation Model, such as that shown below.

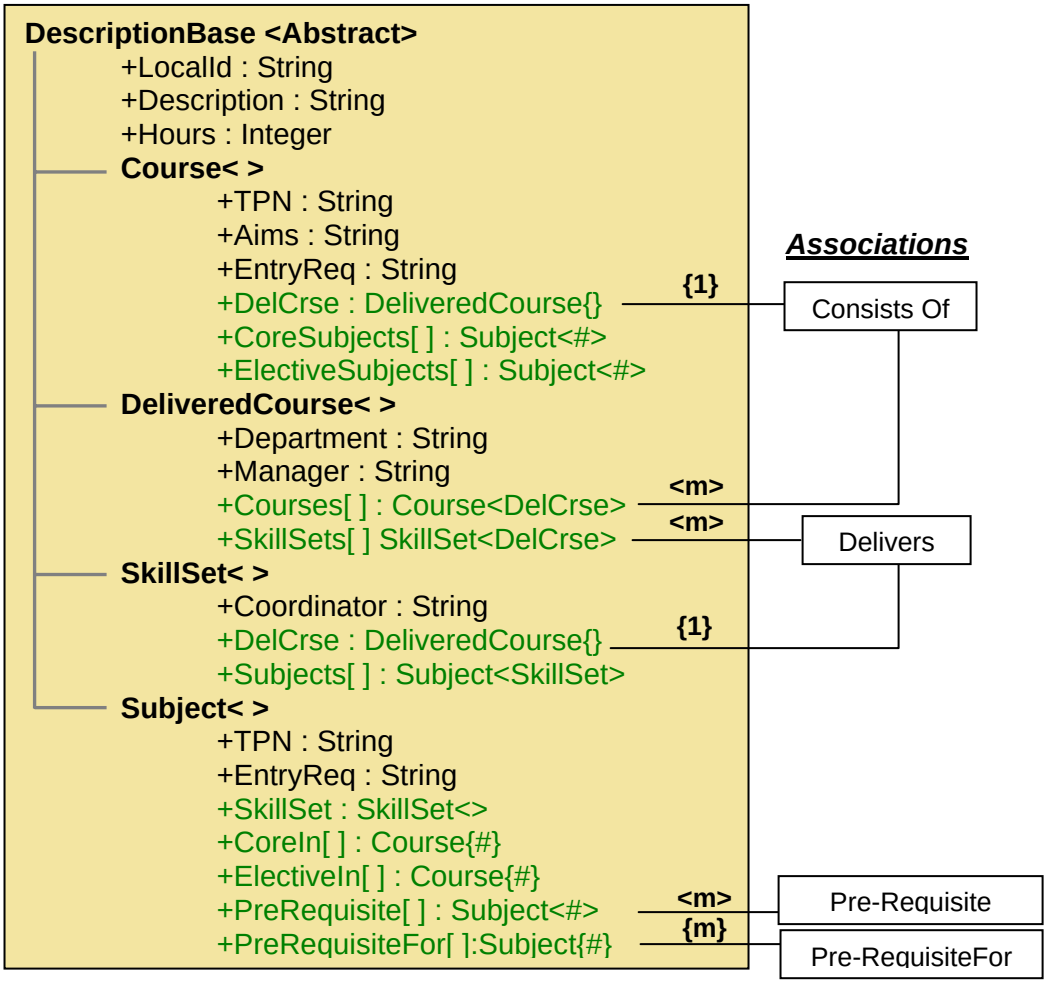


Implementation Model 1 (Using Index and Foreign Key Mapping)

Once the Implementation model has been established, a useful technique is to represent the design as a text-based file. A text-based file has a small foot-print and requires no special graphical support for editing or display, and facilitates the massaging the inheritance tree because the object class data structures and evolving methods are neatly grouped within the inheritance hierarchy.

A text-based Implementation Model, such that for Implementation Model 1 as shown in the next figure, contains all the information represented in the graphical UML styled Implementation Model. It is a very compact representation, allowing even quite complex

designs to be represented on an A4 page. Furthermore, it is easy to maintain and modify with any text editor, and is much closer to our destination, the code.



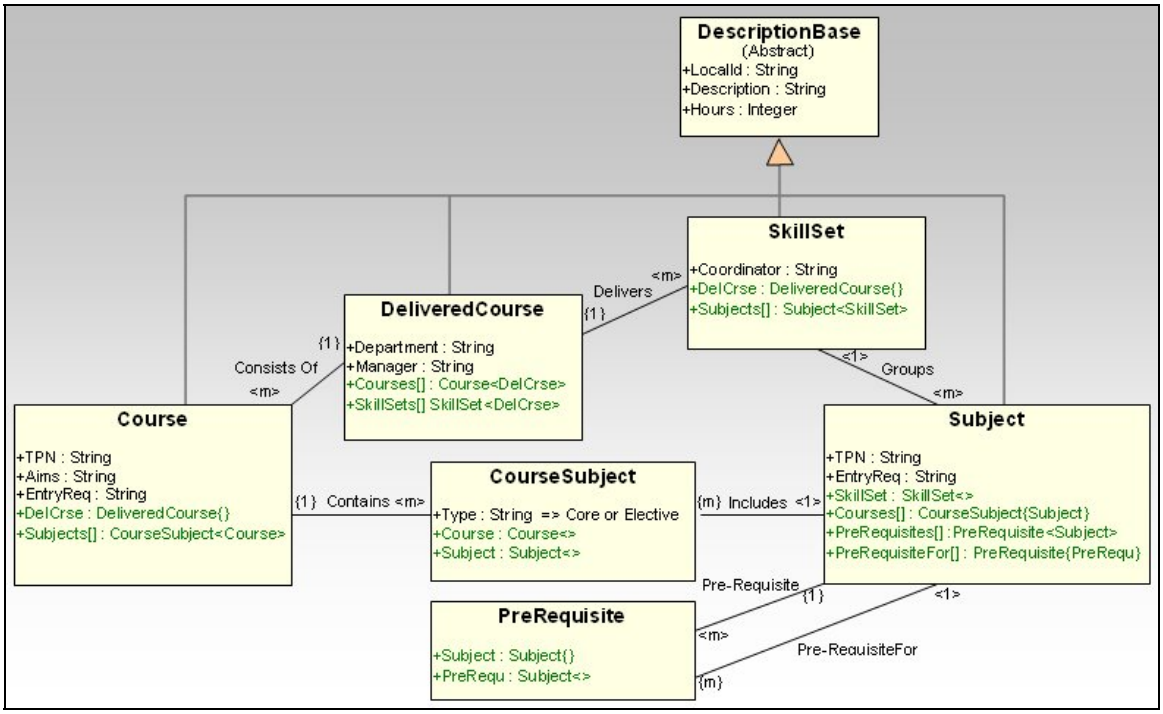
Text-Based version of Implementation Model 1

The treeing has been provided in the text-based example above to highlight the Object Class hierarchy. Method headers can be added after the data structures (properties and private data) declarations using the same level of indentation. Pseudocode logic for the methods is indented under the method headers.

Implementation Model 1 assumes the availability of Index mapping support, whereas Implementation Model 2, as shown in the next figure, relies entirely upon Foreign-Key mapping. It introduces two additional associative object classes - **PreRequisite** to support the reflexive m:m **Subject** associations, and **CourseSubject**.

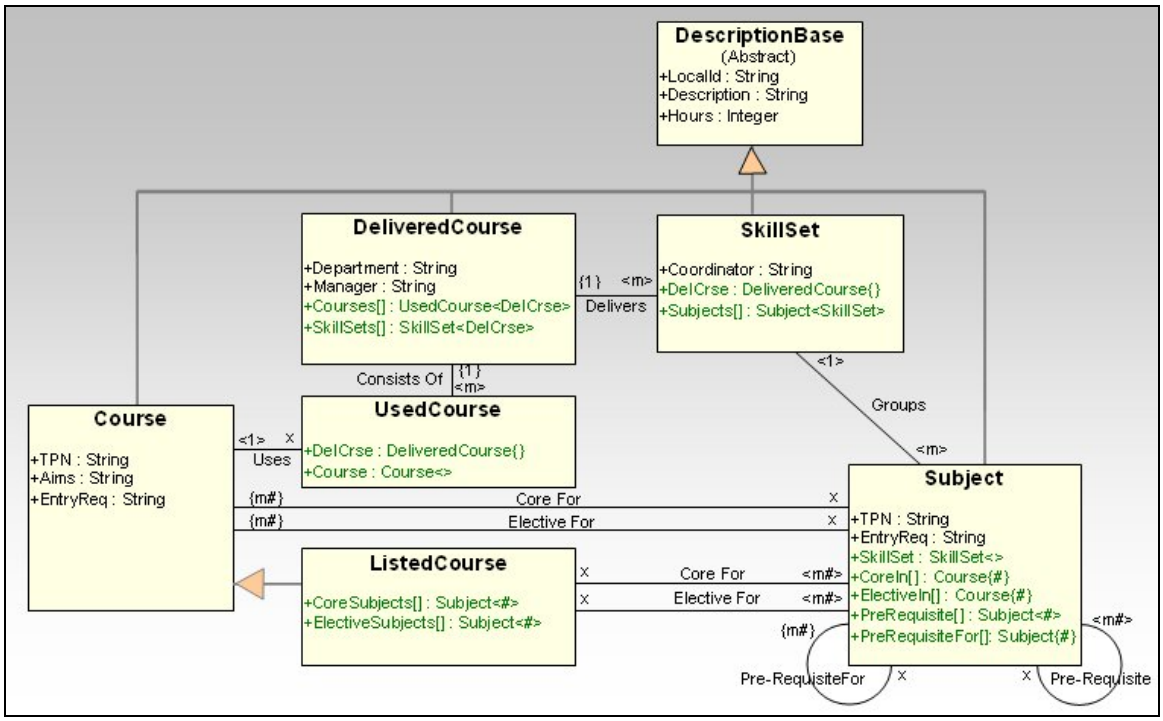
For even such a small-scale application as that represented by Implementation Model 2, the increased complexity created simply by not using Index mapping is quite apparent. As well as being visually more complex, the increased number of associative object

classes, and association support data structures within object classes, will contribute to more persistence overheads and less efficient processing.



Implementation Model 2 (Using Foreign Key Mapping only)

Caveat emptor (user beware) : *Innovative, technically correct designs do not necessarily translate into good implementations*. Below is an example of an Implementation Model that uses a variety of techniques to implement the logical design. Implementation Model 3 is fine in theory, making use of a range of techniques. However, in practice it would result in inefficient processing and produce applications that are difficult to maintain.



Implementation Model 3 (Caveat Emptor)

The Course Information Example: Coding from the Implementation Model

Once the implementation patterns have been decided, the supporting data structures and methods need to be added to the object classes before they can be used in the application.

Before the object class code is created, a careful check should be made of the Implementation Model to ensure that there is a support data structure in the corresponding object class for each side of each association (except for multiplicity **x**).

With an O-R development support package such as ObServer, both the object class code and the object database are generated automatically (you just select the programming language and relational database package) from an implementation design. An extract of the code generated for the object class **SkillSet** is shown below for VB.Net. Business Rule related validation and any encoding logic is placed in the ValidateProperty routine.

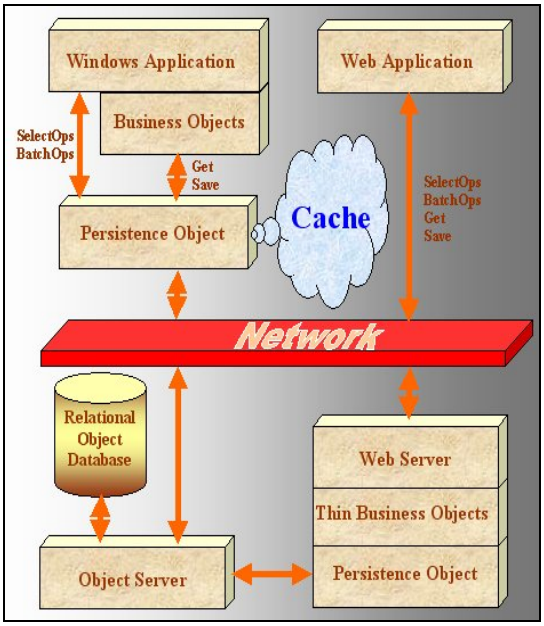
```
' ObServer supported Class SkillSet (ex project Course Information Application)
Option Strict Off
Option Explicit On
Imports ObServerClient, ObServerClient.ObServerPersist

Public Class SkillSet
    Inherits DescriptionBase

Object Private/Protected Variables

#Region "Object Property Get/Set code"
    Public Property Coordinator() As String
        ' > Property No. 4  ............ Design Definition = +Coordinator: String*20
        Get
            Return gObServerPersist.GetProperty(4, Me)
        End Get
        Set(ByVal Value As String)
            ValidateProperty(4, Value, Me)
        End Set
    End Property ' Coordinator

    Public Property DelCrse() As String
        ' > Property No. 5  ............ Design Definition = +DelCrse : DeliveredCourse{}
        Get
            Return gObServerPersist.GetProperty(5, Me)
        End Get
        Set(ByVal Value As String)
            ValidateProperty(5, Value, Me)
        End Set
    End Property ' DelCrse

    Public ReadOnly Property Subjects() As ObjectGroup
        ' > Property No. 6  ............ Design Definition = +Subjects[] : Subject<SkillSet>
        Get
            Return gObServerPersist.GetProperty(6, Me)
        End Get
    End Property ' Subjects

#End Region

Object Property Validation Code

Object Method code

End Class
```

Code Generated for SkillSet Object Class

7 PERSISTENCE SUPPORT OVERVIEW

So far an implementation-planning notation has been introduced, and the data structures required to support the options chosen discussed. This section looks at the architecture requirements to efficiently and transparently support persistence over a network.

The diagram below shows typical 3-tier deployment for Windows and Web applications.



All SQL and stored procedure calls to access the relational database should be placed in a Persistence Object.

For Windows applications the Persistence Object can be located on each client PC, or be shared over a LAN. The object cache should be created and managed in the Persistence Object's address space.

Direct calls to the Persistence Object to select, lock and/or batch-update groups of objects should be available from the Windows application.

Web applications use Thin Business Objects, with Business and Persistence object support on the Server-side. Server-side caching is via the Persistence Object, but Client-side caching tends to be less sophisticated, with thin object data being stored as required in data structures and controls within the Web application itself.

While the above diagram shows typical extremes, many variations are possible. To maximise deployment flexibility and application scalability, the Object Server should only be accessed via the Persistence Object, which is separate to the Business Objects.

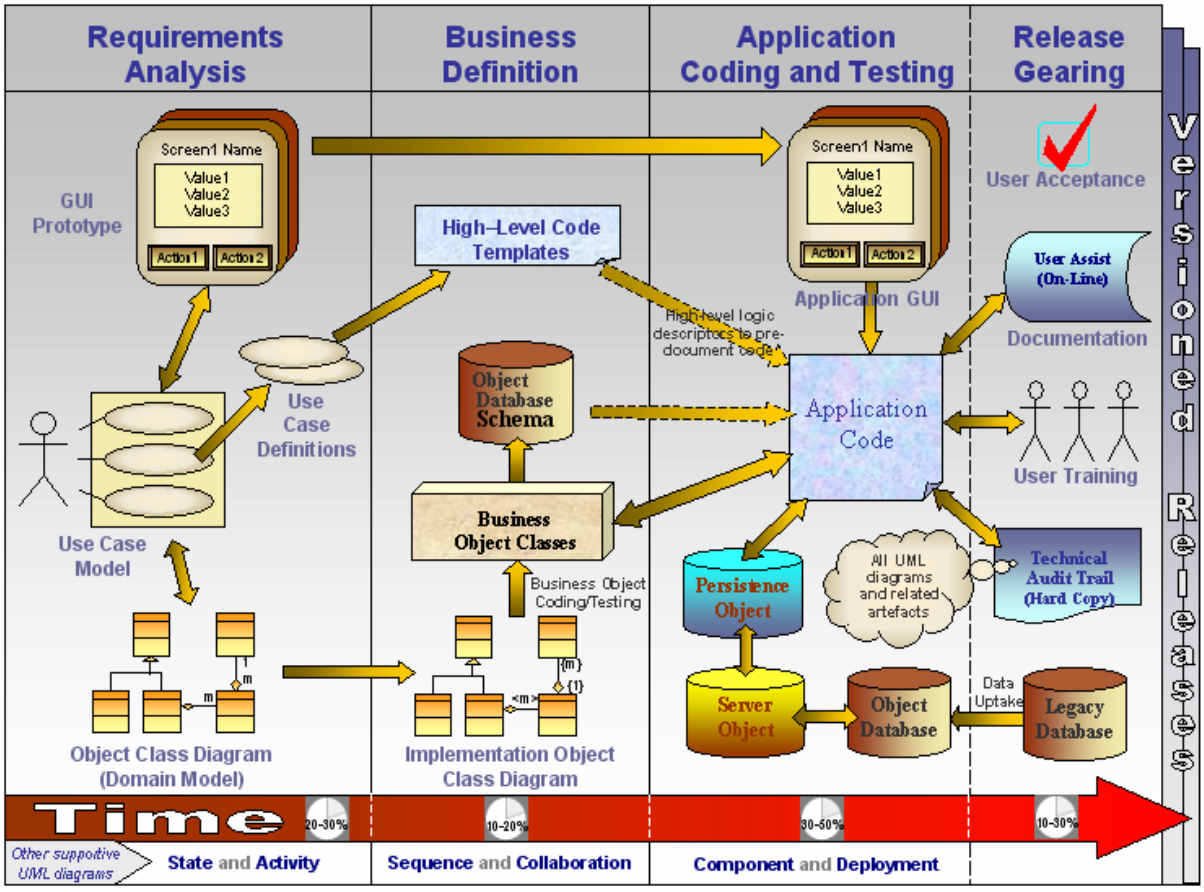
8 BODI BUILDING : AN OO DEVELOPMENT METHODOLOGY

The two most popular Application Development methodologies are RUP (the Rational Unified Process which consists of the 4 phases Inception, Elaboration, Construction and Transition) and MSF (the Microsoft Solutions Framework which consists of the 5 stages Envisioning, Planning, Developing, Stabilizing and Deploying).

Unfortunately neither RUP nor MSF factor in implementation planning. Thus the **B**usiness **O**bject **D**efinition and **I**mplementation (BODI) process has been developed. It consists of four stages (Requirement Analysis, Business Object Definition, Application Coding and Testing, and Release Gearing), with a separate pass is required for each versioned release of an application. The table below compares the three approaches.

MSF	RUP	BODI
Envisioning	Inception	Requirement Analysis
Planning	Elaboration	Business Object Definition
Developing and Stabilizing	Construction	Application Coding and Testing
Deploying	Transition	Release Gearing

The diagram below shows how the BODI building process integrates elements of UML design into a coherent, practical Object Oriented Application Development methodology.



Object Oriented Application Development Cycle

Requirement Analysis involves the parallel, three-way, interactive development of the Use Case diagram, the User Interface prototype and the Object Class diagram to :

- Identify the main processing requirements for each User interface.
- Build user interface prototypes and basic application navigation.
- Identify object classes for the application and their navigable associations.

The output for the Requirement Analysis stage is:

- UML Object Class Diagram (variously referred to as the logical or Domain Model).
- UML Use Case Diagram.
- Use Case Definitions for each Use Case.
- Business Rules approved and signed off by the user.
- GUI Prototype for the application approved and signed off by the user.
- A prototype, preferably signed off by the user.

The **Business Object Definition** stage involves the following tasks:

- Develop the Implementation Object Class Model.
- Code and test Object Class code
- Generate the object database schema.
- Create a pre-documented Application Shell in readiness for Coding and Testing by :

1. Creating a new copy of the signed-off the GUI prototype,
2. Removing any quick-and-nasty prototype code, and
3. Copy-and pasting pseudocode logic from Use Case definitions to pre-document the application's Event Handlers and so act as a guide for coding.

Once Business Object Definition tasks have been completed, coding and testing of the application can commence with a high expectation of success.

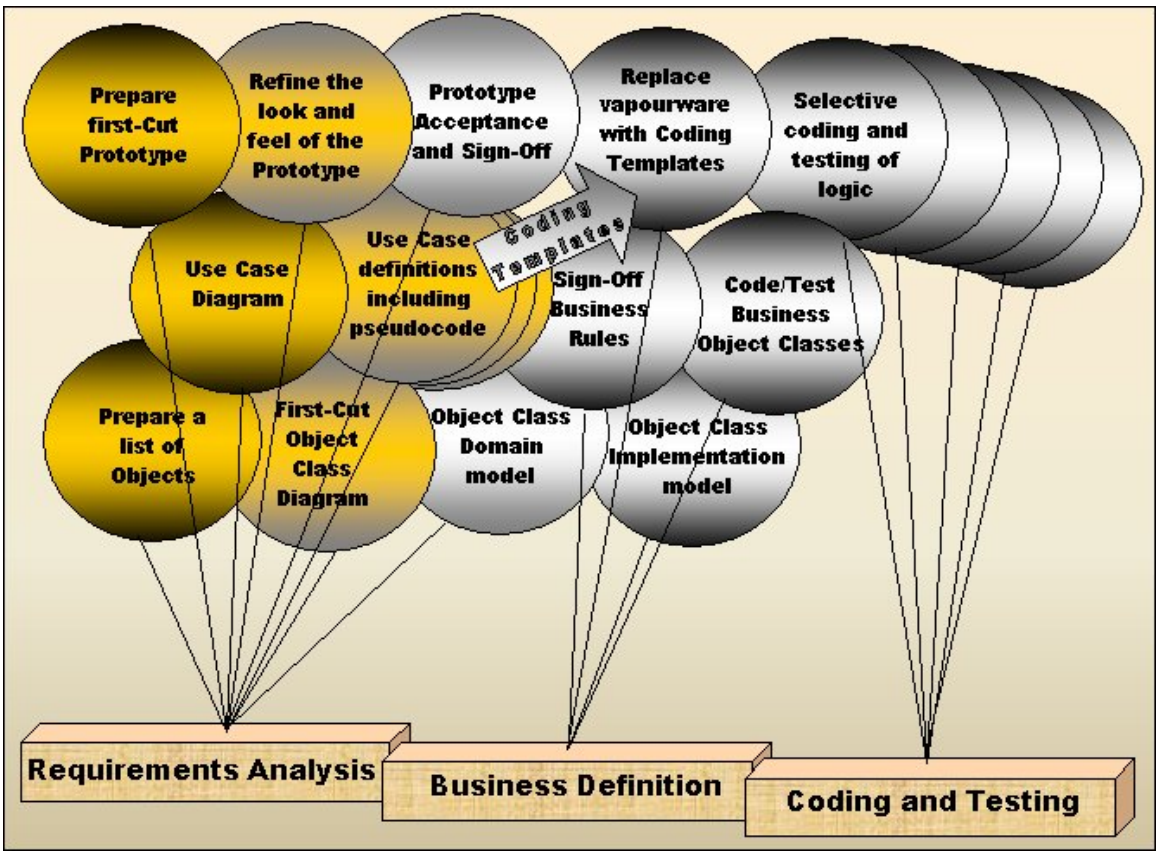
9 BODI BUILDING : ADDING AGILITY

Agile application development involves a series of small-scale, executable-focused activities to build an evolving application. It is a "testing first, immediate execution, racing down the chain from analysis to implementation in short cycles" (Stephen Mellor 2003) approach to application development.

The Agile approach places emphasis upon keeping the development ball rolling without becoming bogged down in a quagmire of design detail. Although executable focussed, Agile developers need enough design activities to be able to develop business object classes, to improve the user interface (the all-important evolving executable), and to generate some other deliverables such as user documentation and training resources.

Agile application development needs an operational context that can provide interaction between agile design and code activities without stifling the creative process. The diagram below represents one such operational context. It shows inter-related agile development activities as balloons arranged in three loose levels. Each balloon is

anchored to a particular stage of the BODI Object Oriented application building process covered in the previous Section.



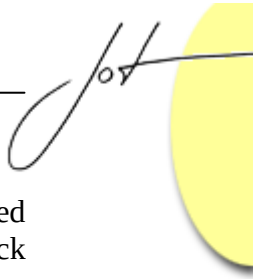
Operational Context for Agile Development Activities

The only two rules that need apply are that you move from left to right within a level of balloons, and that all balloons attached to a particular stage must be completed (or popped) before moving on to another balloon in another stage.

The lower level balloons represent the development history of the Object Class design.

The top level of balloons represents the life-cycle of the prototype to application. For the agile purists, it is the executable "chain moving from analysis to implementation in short cycles", supported by and supporting other design-related agile development activities. For design-led application development proponents, it provides more agility and focus than they would expect from within an object oriented application development lifecycle.

The balloon labelled "Replace vapourware with Coding Templates" is a major transformation point for the executable. By this stage the prototype executable would have served its purpose, which is to develop and prove the application concept and to help advance other agile development activities. Only the static elements providing the



look and feel of the GUI shell of the prototype, as signed-off by the client, are carried forward. The user interface event handlers are, in effect, gutted and refurbished. All quick and nasty prototype vapourware code is removed, being replaced with coding templates derived from activity diagram or pseudocode logic as identified in Use Case definitions.

Coding templates are high-level (as opposed to statement-by-statement code level) logic descriptors or pseudocode, which can be cut-and-pasted into the prototype as comments to document the application and to act as a guide for coders. They will eventually become headers for sections of code or be recycled as code statements (nothing is wasted). And because they refer to business and user interface object names, properties and methods, they provide a far better guideline to coders than the (fr)agile prototype code.

There is no fixed order for the development activities, and the operational context is not overtly prescriptive. Other balloons representing other tasks (eg. the development of state and collaboration diagrams) can be added in for specific projects as required. The Operational Context and BODI Development methodology are complementary, with BODI offering consistency between applications, and the other offering flexibility.

10 SUMMARY

Associations are implemented by the addition of properties and/or methods to object classes. However, because the associated object(s) represent an extension of an object's data structure, it is preferable to define them as properties rather than methods.

For medium to heavy usage associations, cached retrieval supporting the pre-loading of object data should be used for Windows applications. For networked applications, cached retrieval should be mandatory.

With caching comes access granularity (frequency versus volume of access), which can have a major effect on the performance of networked applications. By allowing the access depth of dependent objects to be changed dynamically under program control, access granularity can be adjusted to suit specific retrieval requirements and to tune applications to particular load conditions.

Association implementation planning is essential for the development of efficient object persistence. It requires A FIN analysis to classify the expected frequency of use of each association in each direction as 'frequent', 'infrequent' or 'never'. These classifications are reflected in the Implementation diagram by the < > notation for frequent associations and { } for infrequent associations, to be coded as object and ObjectId properties respectively, or groups thereof.

Assuming the availability of cached retrieval with both Foreign-Key and index mapping, in the absence of other performance factors, it is recommended that Foreign-Key mapping be used for **1:m** associations, and that index mapping be used for **m:m** (where use of an associated object class is not appropriate) and **m:x** associations.

The approach to application development presented in this article emphasises the development of the Logical and Implementation Object Class Models together with the evolving prototype executables. Application development, up to and including the Application Coding and Testing stage of the BODI Application Development process (see Section 8), reduces to three simple steps:

1. **Requirements Analysis** : Develop the Object Class design, and then add annotation and data structures to create an Implementation Object Class Model.
2. **Business Definition** : Generate the Business Objects and Object Database for object persistence. Some commercial packages automate this process, requiring only the addition of logic for application-specific business rules.
3. **Code and Test** the application.

This 3-step approach allows tight coding specifications referring to business and screen objects to be available soon after the Requirements Analysis, and certainly by the end the Business Definition step. Apart from the increased speed of development, the early availability of such specifications provides considerable coordination and control advantages for team-based application development and code outsourcing.

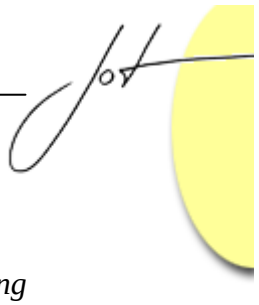
Agile development, while potentially freeing the code-acholic from the shackles of design, can become a nightmare if there is no recognition that some agile code can be quite fragile code. In particular, agile code in the prototype, used to develop and prove the concept to the client, is often poorly structured, poorly documented, quick-and-nasty code. Such prototype code should be jettisoned, leaving behind the GUI shell ready to receive the application code based upon efficient, well-designed business objects.

The operational context described provides the developer with *agility* by providing a number of small inter-related targeted tasks, and *flexibility* in terms of task sequencing.

O-R based persistence objects should represent a high level of abstraction that encapsulates the underlying relational support logic. Relational-specific application code involving datasets, record sets, tables, table joins, rows, or columns should not be required to access and/or to update object data. It is persistence without sequel (SQL), with the closest that application developers need to come to using SQL being SQL-styled requests to retrieve groups of objects (e.g. Get Client where Client.Balance > 200).

With the appropriate O-R support software, there is no need to map objects to a RDBMS, produce code to support database access, or to become involved with relational database issues that can be detrimental to the OO design process. It allows application developers to concentrate upon the retrieval, display and CRUD processing of business objects (individually or as groups) without the need for additional relational-specific code.

As an added bonus, the persistence object's high level of abstraction provides a hedge against technology change, allowing easier transfer to database technologies of the future.



REFERENCES

[Agarwal95] Shailesh Agarwal, Christopher Keene, and Arthur Keller: *Architecting Object Applications for High Performance with Relational Databases*, http://www-db.stanford.edu/pub/keller/1995/high-perf.pdf

[Ambler98]Scott Ambler: *Building Object Applications That Work – Your Step-by-Step Handbook for Developing Robust Systems with Object Technology*, New York, SIGS Books/Cambridge University Press, 1998

[Ambler00a] Scott Ambler: *Mapping Objects to Relational Databases*, White Paper, July 2000, http://www.ambysoft.com/mappingobjects.html

[Ambler00b] Scott Ambler: *The Design of a Robust Persistence Layer for Relational Databases*, White Paper, Nov 2000, http://www.ambysoft.com/persistencelayer.pdf

[Bellware04] Bellware, Scott: *15 Seconds : Object-Relational Persistence for .NET*, *15 Seconds*, January 2004, http://www.15seconds.com/Issue/040112.htm

[Booch et al.00] Booch G, Rumbaugh B,Jacobson I: *The Unified Modeling Language User Guide*, Addison –Wesley, Massachusetts, April 2000.

[Booch96] Grady Booch, *Object-oriented Analysis and Design with Applications*, 2nd ed., The Benjamin/Cummings Publishing Company, 1996.

[Chou et al.00] Shih-Chien Chou and Jen-Yen Jason Chen, "Process Program Development Based on UML and Action Cases, Part 1: the Model", in *Journal of Object-Oriented Programming*, vol. 13, no. 2, pp. 21-27, 2000.

[Faure et al.93] Dennis de Champeaux, Douglas Lea, and Penelope Faure, *Object-Oriented System Development*, Addison Wesley, Harlow, England, 1993.

[Jacobson et al.99] Jacobson I, Booch G., Rumbaugh J. *The Unified Software Development Process*, Addison Wesley, Harlow, England, 1999

[Johnson96] David Johnson, *Programming by Design*, Prentice Hall, 1996.

[Meyer97] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997.

About the author

David Johnson is a Lecturer in the Information Technology Section at the Joondalup Campus of the West Coast College of TAFE, Perth, West Australia. David teaches UML design and Object Oriented Application development. David's interest in Object Oriented design and programming started in 1994 when he conducted an industry survey for TAFE WA to determine future IT skill requirements for TAFE graduates. Over the past five years he has been involved in the development of Design SCOPE's ObServer system. David's E-Mail address is : Johnsd@west_coast.training.wa.gov.au