COPING WITH EVOLUTION IN INFORMATION SYSTEMS – A DATABASE PERSPECTIVE

by

GREGORY LAWRENCE

SUMMARY:

Business organisations today are faced with the complex problem of dealing with evolution in their software information systems. This effectively concerns the accommodation and facilitation of change, in terms of both changing user requirements and changing technological requirements. An approach that uses the software development life-cycle as a vehicle to study the problem of evolution is adopted. This involves the stages of requirements analysis, system specification, design, implementation, and finally operation and maintenance. The problem of evolution is one requiring proactive as well as reactive solutions for any given application domain. Measuring evolvability in conceptual models and the specification of changing requirements are considered. However, even "best designs" are limited in dealing with unanticipated evolution, and require implementation phase paradigms that can facilitate an evolution correctly (semantic integrity), efficiently (minimal disruption of services) and consistently (all affected parts are consistent following the change). These are also discussed.

KEY TERMS:

Software evolution; Requirements change; Schema evolution; Conceptual Modelling; Meta-modelling; Model-Driven Architecture; Orthogonal Persistence; Persistent Application System; Evolvability; Software development life-cycle; Information System design and development; Database design; Database evolution; Application evolution; Unanticipated change

COPING WITH EVOLUTION IN INFORMATION SYSTEMS – A DATABASE PERSPECTIVE

by

GREGORY LAWRENCE

submitted in part fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

in the subject

INFORMATION SYSTEMS

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROF K.V. RENAUD

NOVEMBER 2002

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION

Business organisations today are faced with the complex problem of dealing with evolution in their software information systems. This effectively concerns the accommodation and facilitation of change, in terms of both changing user requirements and changing technological requirements. Information systems in this context refers to any system having to manage the storing, structuring and processing of data. This research is particularly concerned with the database (persistent data) aspect of information systems and regards this as a core element that must be addressed in terms of the evolution problem.

Evolution affects both structural and behavioural elements of the persistent store, or database. Any changes to the database will in turn influence applications dependent on the database schema – an aspect usually considered as stable and static in terms of structure and behaviour. However, change is inevitable, stemming from both user and organisational sources. The following trends define the nature of these evolution-related pressures:

- *Integration pressures*: Organisational mergers often require conversion and adaptation of database systems. However, the investment in persistent data is often considerable, requiring that any change not disrupt the integrity and consistency of persistent stores. The field of schema evolution is particularly relevant here and organisational dependencies on so-called "legacy systems" should not be underestimated.
- *Changing domain requirements*: Evolution stemming from changing requirements in the underlying application domain (or Universe of Discourse (UoD)) is inevitable. This takes the form of changes in product lines, government regulations, or other organisational standards relating to particular domain entities or the inter-relationships and inter-actions between them.
- *Containment of software maintenance costs*: Frequent software maintenance is expensive, both in terms of disruption to system availability and services, as well as increases in the complexity of the system. The latter can result in systems being difficult to comprehend by maintainers whom are not necessarily the same individuals that were responsible for designing the initial system. More efficient and effective maintenance mechanisms are sought. However, the ability to

accommodate evolution without resorting to any significant system maintenance is desirable.

Technologically-oriented pressures also contribute to evolvability requirements. These include:

- *Personalised (Adaptive) software*: The notion of software that dynamically evolves to suit its current environment, or context, is prevalent in software applications ranging from word-processors to e-commerce. An example of the latter would be the personalisation of product prices and discounts, depending on the customer, time of year, etc. From an information systems point of view, this requires consideration regarding the design of software that will model the core or fundamental aspects of the domain, as well as cater for the personalisation requirements and policies that may change over time. This has traditionally been a difficult problem, leading to software that ultimately becomes very difficult to maintain.
- *New application classes*: More dynamic schemas and behaviour are required to manage the requirements stemming from application areas such as data mining and scientific databases modelling evolving entities (e.g. weather systems, biological systems). Here, data structures must evolve as they are discovered. This is clearly in conflict with traditional database development where schemas exist before any programs are run against them.
- *Platform evolution*: The recent proliferation of different middleware platforms such as .NET, EJB and CORBA, has made it difficult for enterprises to standardise on as single platform. Frequent platform changes are also disruptive to the organisation, resulting in loss of investment in certain implementation technology. Pressure for more conceptually-oriented development, that is also platform-independent, is increasing.

The range of requirements presented here necessitates a consideration towards the way information systems, and their persistent stores in particular, are conceptualised, designed and finally implemented. An approach that uses the software development life-cycle (SDLC) as a vehicle to study the problem of evolution is therefore adopted.

This involves the stages of requirements analysis, system specification, design, implementation, and finally operation and maintenance. The maintenance phase is of particular interest to long-lived systems and especially to evolution-oriented requirements changes. These often necessitate a re-iteration through earlier phases and are notoriously expensive in terms of cost and disruption to system services.

We now briefly discuss the content of this research in terms of the remaining chapters.

Chapter 2 describes the stages of the SDLC with emphasis on the database aspect. Conceptual modelling as a means for describing the entities and relationships comprising an application domain is considered, as well as the issue of describing data not traditionally stored in DBMSs. In particular, the Extensible Mark-up Language (XML) is regarded as a means for describing data not easily characterised in terms of rigid schema structures.

Having introduced the processes and artefacts involved in constructing, implementing and maintaining information systems, Chapter 3 proceeds to discuss the impact of evolution on the SDLC phases. The characteristics of traditional software development that affect evolvability are also discussed. These include:

- The need for requirements analysis techniques that are better coupled and integrated with specifications so as to preserve consistency through numerous maintenance iterations.
- The need to quantify and qualify evolvability in conceptual models (specification phase) and the need to model changing requirements.
- The problem of design erosion as it affects the architectural design of the system.
- The schema evolution issue and the associated problems of application and database compatibility. Weaknesses in current implementation technology are also considered. The problem of using disparate components (i.e. database systems, operating systems, communication systems, etc.) to realise an implementation is of particular concern.

Chapter 4 introduces a range of techniques presented in the literature as solutions to particular aspects of the evolution problem. These are once again discussed in terms of the SDLC phases and include the following:

- The issue of stability and how it may be assessed in conceptual models.
- Approaches to deal with the specification of evolving requirements – both in terms of structural and behavioural models.
- The issue of accommodating evolvability into design-level artefacts. In particular, the *interactions* between design components emerges as a fundamental aspect in coping with evolution.
- Solutions to the schema evolution problem are considered for their contribution towards facilitating maintainers with the means to evolve persistent stores whilst preserving semantic integrity.
- Reflection, being '*the ability of a program to manipulate as data something representing the state of the program during its own execution*' [Gabriel et al, 1993], is considered as a means for facilitating dynamic behavioural evolution.
- Persistent application systems and orthogonal persistence as a programming paradigm where developers are freed from the concerns of implementing explicit mappings and translations between disparate system components. For example, the impedance mismatch problem between programming languages and database query languages is a result of such "incompatibility".
- A meta-modelling architecture as a means for realising high-level semantic compatibility between system components. In particular, the OMG's Model-Driven Architecture (MDA) is presented as an approach where development is more specification-level and conceptually oriented. It uses Unified Modelling Language (UML) models that are mappable to platform specific implementation artefacts. This mapping is, for the most part, automated via tools. The benefits for managing evolution are considerable, particularly the assurance of consistency and controlled change propagation via a "meta-conforming" system model.

Lastly, Chapter 5 provides a synopsis on the problem of evolution and its solutions. Section 5.2 briefly discusses the requirement of providing a framework to support evolution uniformly through the software development life-cycle.

CHAPTER 2 - DESIGN AND DEVELOPMENT OF INFORMATION SYSTEMS

The development of an information system typically follows the phases of the software development life-cycle. These include a requirements analysis phase, followed by a system specification phase, design phase, implementation and testing phase, and finally an ongoing maintenance phase.

Numerous techniques can be applied at the early stages of requirements analysis and specification in order to capture and model customer requirements for a particular application domain. These include:

- The modelling of the entities and their relationships in the application domain. In particular, this refers to the database aspect where conceptual modelling provides the means to formally describe the real-world objects, their roles and inter-relationships, as well as constraints or business rules.
- Functional requirements (including those relating to the process and data flow in a system that define the transformations expected by the user)
- Non-functional requirements (cost, reliability, availability and performance)

The design phase in an information system development includes design of the overall system architecture where the components making up the system are described, as well as the relationships between them. This phase bridges the specification and implementation phases by considering *how* the specification might be realised as opposed to *what* it must realise.

The implementation phase is the realisation of the design in terms of a set of programs or program units as well as the instantiation of a physical database schema. Testing now occurs to ensure that the system meets its specification.

The maintenance phase is a particularly interesting and challenging one for long-lived information systems. It concerns the sustainability of the system in the face of changing requirements, changing technology, or the correction and improvement of

flaws. Through all this, the developer is required to maintain system integrity and consistency with minimum disruption to services.

We now consider each stage of the development life-cycle in more detail, particularly with a view towards information systems where the persistence of data and its semantics must be addressed.
Sections 2.1.1 through 2.1.5 consider requirements analysis, system specification, design, implementation and testing, and the maintenance phase in further detail. Section 2.2 presents an introduction to conceptual modelling – a key component for formally describing the database aspect of an information system. XML as a means to model Web, and other unstructured data, is introduced in section 2.2.2.

2.1 The Software Development Life Cycle

2.1.1 Requirements Analysis

This phase is characterised by consultation with the client in order to determine the system's services, constraints and goals. The problems for which a solution is sought are documented so as to create an initial requirements definition. Sommerville [Sommerville 1992] suggests that although requirements are indeed stipulated at varying levels of abstraction, the initial liaison with the client should typically be written in a natural language style that is understandable by all stakeholders. Simple intuitive diagrams are also prevalent at initial meetings, particularly for indicating the major components of the system, their inter-relationships, as well as the relationship to the greater environment within which the system operates. A general recommendation is to also include a form of definition that is susceptible to logical reasoning. This is required to facilitate an effective transition to the next phase - system specification. The requirements analysis stage tends to be iterative in the sense that better understanding and re-evaluation of the problem domain often result in revised documents. The extent of such "improvement" is naturally capped by the designer's or client's foresight at that time.

In addition, non-functional requirements such as restrictions on the freedom of the designer relating to cost, hardware limitations, resources, etc., should be expressed during this period.

The phase is usually completed by a requirements definition document that includes the following [Sommerville 1992]:

- *Introduction*: Motivation for the need of the system, and placing it into context as regards the overall business and strategic objectives of the organisation commissioning the software.
- *The system model*: This should indicate the system components and the relationships between them, in addition to the relationships between the system and its environment. High-level abstract data models may be specified as well as simple action diagrams to indicate transformations that occur in the system. The aim is to indicate the real-world entities that are to be represented in the software system.
- *System evolution*: The fundamental assumptions on which the system is based, together with anticipated changes due to hardware evolution and changing user needs, should be presented.
- *Functional requirements*: The services provided for the user should be described in natural language terms. This would also include constraints on the nature of the system's functionality. However, as suggested earlier, a form that will facilitate cross-reference to a more formal specification is advantageous.
- *Non-functional requirements*: Constraints imposed by the environment in which the software must operate should be related to the functional requirements.

In order to progress to a more formalised interpretation of the system requirements, specifications are developed. These serve to bridge the gap between informal requirements analysis and system design.

2.1.2 System specification

This phase concerns the creation of an abstract description of the software. A basis is then established by the system designer for later design and implementation phases. The need for greater formality becomes paramount in order to avoid the ambiguity

and imprecision that is inherently part of informal natural language and informal graphically-based formats.

There are different aspects of an information system to be specified. Broadly speaking, these include means to model the process and flow of control aspects (defining the sequencing of events or their synchronisation), the behavioural aspects, as well as the more structural or entity related concerns. So-called *business rules* should also be crystallised at this point. These essentially specify constraints on the behavioural aspect of the system.

Developing a good specification is a difficult task. In particular this includes finding the right level of abstraction at which to decompose the system components, as well as finding an appropriate specification language or model that is sufficiently expressive. It should be noted that one of the prime objectives of using more formally-oriented specifications is to obtain a basis for sound representation of the system's semantics. An ability to support some form of logical reasoning is therefore established, allowing the specification to be interrogated.
Balzer [Balzer 1986] elaborates on the principles and implications of good specifications. His criteria provide a sound basis for presenting what the specification phase of a system development should entail. These include the following notions:

- Functionality must be separated from implementation in order to bolster the ability to reason *logically* about a specification without being hindered by implementation level constructs. Constraints on the functionality should be specified non-deterministically.
- A means to model process oriented aspects, or stimulus-response type behaviour, is required in order to specify the dynamics of the system. Process algebras are noted as being favourable for formally describing the ordering of events in the system workflow sense.
- Interaction and relationships with other system components, as well as the greater environment, should be modelled. This requires that a globally maintained model be used to capture the system context. He emphasises that a mutual dependence between interacting system parts and the environment can lead to implementation-bias pitfalls. A *demon* capability, as an independent agent that deals with

environmental changes, is therefore recommended. This avoids specifying the methods regarding the interaction between components.

- The system specification must be a cognitive model, in the sense that it corresponds to real-world objects and their actions in the domain as perceived by the user community. It must also aim to incorporate the rules or laws governing these objects, which may constrain certain states of the system, as well as specify how objects respond when acted upon. As a result of this, constraint statements are required. These are in addition to constraints that specify type-checking concerns in the sense that real-world objects are governed by belonging to one or more types.
- An operational specification is advantageous in that it facilitates the validation of an implementation against a specification.
- A specification should be capable of being augmented and dealing with incompleteness. Changes to invariants and underlying declarations should ideally be promulgated throughout the specification.
- Lastly, the specification should be localised and loosely-coupled. Localisation facilitates that only a single piece of the specification requires alteration in the event of a system modification. Loose-coupledness facilitates the addition and removal of pieces of a specification.

As suggested above, there are several aspects that need to be modelled; the primary ones relating to process control flow, some form of entity and relationship modelling, and behavioural modelling. Above all of this, a *good* specification would aim to integrate these in order to realise the benefits described in [Balzer 1986]. However, numerous factors, mostly pragmatic, also play a major part here.

Firstly, as suggested by Wing [Wing 1990], developments that aim to use formal modelling techniques inevitably face the issue that no one method can satisfy all modelling requirements. Other issues such as available tool support, rapid application development requirements, unfamiliarity with formal notations, and so forth, also add to a system specification that is far less formal than Balzer's [Balzer 1986] notion of a specification.

Information system developments primarily follow a structured analysis type approach of which entity-relationship type modelling is a primary method for

specifying the database component. The focus here is more toward a conceptual model as the primary description of the objects stored in the database (Conceptual modelling approaches are discussed in further detail in section 2.2). Specifying the behavioural aspect of an application, whether it be encapsulated within the constructs of a conceptual model or not, can take a variety of forms ranging from formal methods with sound logical reasoning and mathematically-oriented constructs, to more pseudo-code type approaches with greater implementation bias. The more process-oriented requirements of an application are usually modelled separately, using techniques such as data flow diagrams to capture the activities and events that determine the flow of control.

Overall, some form of modularization takes place during specification, rendering concrete entities of the application into abstract components of the specification. To progress the system's development, implementation-oriented abstractions of the specification are produced. The form and interaction of these more refined components characterise the next phase – design.

2.1.3 Design

This phase is initially characterised by the development of an architectural design. This aims to obtain a more concrete description of the different software components or sub-systems along with their relationships. It must be emphasised that the relationship between specification and design is a close one. Sommerville [Sommerville 1992] notes that '*although the process of setting out a requirements specification as the basis of a contract is a separate activity, formalising that specification may be part of the design process. In practice, the designer iterates between specification and design*'.

For large enterprise developments in particular, the design process can be complicated by the volume and diversity of different components. These components may include those pertaining to the database aspect, components realising workflow-related concerns, as well as user-interface objects. The need to manage and relate these in more abstract terms has given rise to the field of Software Architectures. This has emerged as a means to assist developers at the design phase with a means for

specifying the *overall* system structure in terms of the elements that comprise the structure and the relationships between them. The aim is to provide the developer with a set of intermediate models and artefacts for bridging the gap between specification-type entities and an implementation in the form of program code. As discussed in [Hofmann et al, 1996], coordination between components is conceptually handled by the notion of a *connector*. These "attach" to components via *ports,* which can be regarded as the interface of the component.

For the purposes of this research, the issue to note is that the design phase concerns the macro-level view (software architecture) defining the overall system in terms of the components and the interactions between them, as well as the refinement of individual components' specifications.

There are different design strategies for refining these components. The primary consideration is whether to adopt a functional or object-oriented approach. The former involves a decomposition into functional components with a centralised system state that is shared between the functions operating on that state. With an object-oriented approach, the system is viewed as a collection of objects with decentralised system state where each object manages its own state information. Objects communicate by calling a procedure or method associated with another object. The major consideration regarding design is that there is no "best" approach as such. Software systems are usually designed with a combination of both strategies but in such a way that they are complementary. Sommerville [Sommerville 1992] states that each may be applicable at different stages of the design process. This implicitly includes the specification phase as well, due to the iterative nature of development and specification refinement. It has been suggested that object-oriented techniques are most natural at the highest and lowest levels of system design. In particular, the natural high-level view of a system is as a set of objects, or architectural components. However, when the system is examined in further detail, functional descriptions of certain elements may tend to be more natural. Lower level detail would typically be concerned with manipulating objects, thus requiring an object-oriented approach once again.

In summary, this pivotal phase between abstract specification and implementation is vast in the sense that it deals with the refinement of high-level definitions into low-level implementation-oriented constructs. Through all this, verification of the design against the specification is required in order to ensure correctness, particularly in

terms of semantic integrity, i.e. is the design a correct interpretation of the specification.

Amongst the system components being refined and verified is the database conceptual model - one of the primary aspects of an information system's architectural design. According to Bowers [Bowers 1993], the *conceptual model* is an abstract specification of the data to be included in a database system, while the *conceptual schema* is that same model but cast into a form which can be implemented. The process of transforming from conceptual model to conceptual schema is known as *conceptual schema design*. There are a number of distinct types of database systems, characterised by the principle structure used to represent data; for example, hierarchical, relational, or object-oriented structures. These, in essence, represent the *logical* implementation structure for the database.

The architectural framework and its constituent components are finally refined to a point where they can be implemented.

2.1.4 Implementation and Testing

The implementation aspect involves the translation of the design into a set of executable programs or program units. Any outstanding details concerning functions to be performed and the nature of the data involved, need to be settled in order to allow the phase to reach a conclusion. Unless a prototype development was undertaken, certain requirements and design omissions may only be revealed at this point. Iteration through earlier stages would therefore be required. In addition, non-functional requirements, for example those relating to performance requirements, are typically verified at this stage.

As far as the database aspect is concerned, the logical schema is now implemented using a specific data definition language (DDL) for specifying the required types and structures, as well as any corresponding validity constraints. Bowers [Bowers 1993] notes that it is often the case that semantic and integrity-related features represented in a conceptual model are inadequately portrayed in an implementation, due to omissions or restrictions inherent in the particular DBMS software. Less than perfect workarounds usually result to minimise such shortcomings.

Implementation and testing usually takes place on a unit by unit basis followed by integration and testing of the parts into a whole. Verification against the specification plays a large role here and can be handled differently for different software components, depending on the development process. Developments, by formal transformation from specification to implementation, guarantee that the final implementation is indeed an adequate representation of the specification, while other less formally-oriented approaches rely on an adequate set of test cases to be run against the system. These are then verified against the specification.

2.1.5 System Maintenance

The system maintenance phase is of particular relevance to the problem of evolution: It is usually the longest phase in the life-cycle, especially so for information systems which are largely comprised of long-lived objects.

Boehm's [Boehm 1981] definition of maintenance as *'the process of modifying existing operational software while leaving its primary functions intact'*, succinctly captures the role and intention of typical activities that occur during the maintenance phase. According to Lientz and Swanson [Lientz and Swanson, 1978], these would include the following:

- Corrective maintenance (detecting and correcting errors, i.e. routine debugging)
- Adaptive maintenance (accommodation of changes to the environment of the program - specifically hardware and newer software technologies for the implementation of system units)
- Perfective maintenance (user requested enhancements, improved documentation, enhanced performance)

Lientz and Swanson [Lientz and Swanson, 1980] reported that the respective categories count for 17%, 18% and 65% of the total maintenance activities, and that user requested enhancements, in particular, accounted for two-thirds of the last category. This would also include requirements changes, in the sense of changing business rules, new government regulations, etc., as introduced in Chapter 1.

Software engineering research regarding the study of system change has followed a combination of both theoretical and empirical routes. These include investigations into the relationship of organisational factors to software systems, as well as the

design process used to engineer the software. The former considers both structural issues in organisations, affecting the way in which systems are initially conceptualised, as well as factors influencing the ease with which changes can be realised; for example, the size and complexity of the software, or the effectiveness of the resources allocated to realise the change.
Investigative studies, regarding the engineering techniques used for developing software, have also proved informative. Overall, lifetime costs are generally decreased by an increase in effort during the earlier development phases. Furthermore, techniques such as object-oriented design, approaches encouraging module independence, and use of high-level programming languages, are all considered to be favourable towards aiding maintainability.

As regards databases, the traditional ANSI/SPARC architecture delivers data independence, whereby changes at the physical level do not compromise logical or conceptual schemas, and external or user schemas can be changed without affecting the database's conceptual schema. However, as implied above, requirements change is a serious issue resulting in considerable maintenance overhead - databases are not exempt from this. Methods to cope effectively and efficiently with such change across the information system infrastructure are welcomed – this includes the database and the application structures "surrounding" it. The issue of evolution, and how it might be dealt with, is considered extensively in later chapters. In particular, the effect at each stage of the development life-cycle is discussed, along with the problems that need to be addressed and the solutions presented in the literature. The requirement for addressing evolution coherently and consistently over all stages of the life-cycle, emerges as a distinct requirement for the long-term maintenance of an information system.

As this research is particularly concerned with the persistent (database) aspect of information systems, we briefly survey conceptual modelling techniques. These provide the core representation for the (persistent) information content of an application domain, and are therefore an important reference when addressing evolution over the development life-cycle.

2.2 Conceptual Modelling for Databases

The traditional role of a conceptual model is to provide an implementation-independent representation of the information content of a particular application domain or *Universe of Discourse (UoD).* It serves as a first step for creating a more structured and unambiguous view of real world objects and the roles they play within the application. A more design-oriented interpretation, or conceptual schema, is then established, whereby the permitted states and transitions of a database system are defined. This abstraction should not be concerned with data alone per se, but also about how it is used in order to provide a control for maintaining the integrity and validity of the system. The conceptual layer is, or should at least be, the most stable, unaffected by user-interface changes (external views), or any physical storage and access methods. The notion of conceptual model stability, and hence later conceptual schema stability, as it affects system evolution, is discussed in detail in Chapter 3.

In order to provide a handle on formalising what the data in a database means, much research was initiated in the 1970's and 80's toward semantic modelling. In order to verify user requirements with a customer, analysts required a suitable means of representation of the environment. Secondly, the system designers needed this as a basis for their design of computer systems. Finding a technique that suits both camps sufficiently remains a challenge. In particular, the realisation of this would hold promise for managing user requirements change more consistently through the development life-cycle. Chapter 5 considers this in further detail.

Date [Date 1990] characterises semantic modelling research as approaches in terms of the following steps:

- Attempts to identify a set of semantic concepts that allow informal discussion about the real world. This typically includes the notions of entities (distinguishable objects), properties (a piece of information describing an entity), identity (a property of an entity that serves to identify it), relationships (an entity that interconnects or relates two or more other entities), subtypes and so on.
- A more formal symbolic representation that corresponds to the above mentioned concepts.

- A set of formal integrity rules that constrain a model in such a way that the concepts expressed in it are valid in a meta sense, i.e. that the model is a valid instance of a meta-model.
- A set of formal operators for manipulating the formal objects. These might be applied to construct a set of different user views from one base structure.

Traditional conceptual modelling approaches, such as Chen's basic Entity-Relationship model [Chen 1976] and its later extensions, have been widely used as a mechanism to capture semantic concepts including the notions of entity, property, relationship, and subtype. However, the role of a sound conceptual model is becoming increasingly pervasive where factors such as the semantic expressiveness of the model, ability to serve as a formal specification for a system, understandability by humans, and support for automated mapping to lower level DBMS structures are important. The ability to incorporate a system's behavioural characteristics is also becoming a necessary requirement, in order that both system state and process are modelled in a unified manner.

The following section aims to provide a brief overview of the major conceptual modelling techniques. This should provide the reader with an appreciation of the scope of their role as well as an indication of their future potential for bolstering information system design and management.

2.2.1 An overview of Conceptual Modelling Techniques

The Entity-Relationship (ER) Model

Developed by Chen [Chen 1976] and still widely used today in numerous refined and extended forms. It delivers a set of analogues to the semantic concepts introduced above and also introduces a corresponding diagrammatic representation. Although useful as a basis for abstract database design, it is often cited as lacking in its ability to express integrity related constraints. This includes constraints pertaining to attributes of entities, and those describing the nature of relationships. By not dealing explicitly with constraints, instances of the ER model are vulnerable to mis-interpretation which can have far reaching effects when design transformations need to be applied.

Nonetheless it has served well as a simple and readily understood means for communicating the salient features of a particular database design.

Object-Role Modelling (ORM)

Using Natural Language Information Analysis Method (NIAM) as a basis, Halpin [Halpin 1995] specified the Object-Role Modelling (ORM) method as a technique that delivers an improved means for describing business rules and constraints. In particular, the exact relationship of an attribute to its entity is more explicitly modelled. Underlying domains, relationship cardinalities and optionalities, and an ability to be easily populated with real world instances, all help facilitate easier validation with users in natural language, in addition to providing a means that is more feasible to formulating, transforming, or evolving a design. Halpin [Halpin 1995] indicates that the model has proved suitable as a conceptual basis for both relational and object-oriented data models. The latter is facilitated through better support for subtype and inheritance representation than that provided by the ER approach.

He also states that ER diagrams can be abstracted from ORM diagrams, providing the means to present more compact and focused summaries.

Object-Oriented Modelling

Object-oriented database design originated from the approach introduced by object-oriented programming languages, where the user need not deal with computer-oriented constructs such as records and fields, but rather with objects and operations on those objects, thus resembling real world counterparts more closely. Date [Date 1990] notes that object-oriented technology is not a semantic modelling technique per se, as the latter aims primarily to identify a set of constructs at a higher level of abstraction that are generically useful and which recur in some shape or form in a wide variety of applications. Halpin [Halpin 1995] mentions the object-oriented database approach as tending to be a mix of conceptual, logical and internal elements. The development of UML (Unified Modelling Language) is also providing a standard

notation for representing the structure of data in the object-oriented community. Its role in the OMG's Model Driven Architecture [OMG-Soley, 2000] as a means for managing evolution is particularly important. We discuss the MDA in detail in Chapter 4. At this juncture, a brief discussion of object-oriented modelling as a role player in conceptual specification is warranted.

The principal terms and concepts of the object-oriented approach include object itself, class, method and class hierarchy. Every object has a unique object ID and may be as simple or as complex as required. Complex objects can be constructed from combinations of existing objects which can in turn be simple or complex. Whereas objects more or less correspond to the notion of variable in the programming language sense, classes correspond to type, or more appropriately, abstract data type. Classes therefore group objects based on common characteristics. These characteristics include attributes as well as methods. Methods are essentially operators that apply to objects and therefore add to the ability of this notion of object to model both state and behaviour of real-world objects. These methods are incorporated within the "public interface" for objects of a particular class, while the detail of their implementation is hidden or encapsulated from the user. Classes can also be grouped into superclasses based on some set of common characteristics among the subclasses. Each subclass in this hierarchy inherits the common set of attributes and methods from the superclass. Generally, subclasses can modify or override inherited characteristics, as well as add additional ones. A class hierarchy can also join with other hierarchies at a particular level, thereby allowing a class to inherit from multiple superclasses. As indicated in [Bukhres et al, 1996], the inheritance mechanism can prove useful for abstraction and polymorphism. Upper layers represent more generic and abstract views of lower-layer characteristics. Software reuse is also encouraged as subclasses are allowed to use code and storage structures defined in ancestor classes. Polymorphism in the object-oriented sense applies to references (relationships or method invocations) that can refer to objects from multiple classes. The context of a reference can change over time as application needs change. For example, an application calculating the area of a shape for some object, will inherit the attribute of "enclosed area" and method of "surface area" from some more abstract class of geometric shape but, depending on the context, will utilise the particular method applying to square objects, round objects, or say triangular objects.

As mentioned earlier, UML is emerging as a standard notation for object-oriented modelling. This would seem promising for a start as the object-oriented data model has been plagued by the problem of not having an exact definition. UML presents a framework where *classes of entity objects* are essentially entities, and *associations* are relationships. Hay [Hay 1999] notes its more extensive capability to describe inter-relationship constraints. Furthermore, it adds the ability to describe the behaviour of each object class - usually in the form of pseudo-code or C++. In summary, it attempts to provide a formalism suitable for both requirements analysis and design. However, some implementation level concerns creep into the notation and are sometimes criticised for being distracting and unnecessary from a conceptual point of view.

The surge of Web-based and semi-structured data also requires consideration, especially as it is inherently prone to evolution. We now consider the Extensible Markup Language (XML) as a means to facilitate description of such data.

2.2.2 XML - Describing data on the World-Wide Web

As Tanaka et al. [Tanaka et al, 2000] state, there is also a need for treatment of data not stored in traditional DBMSs. In particular, this includes semi-structured data which is prevalent on the World-Wide Web and typified as being difficult to describe in terms of rigid schema structures. Techniques are therefore sought to deal with data that is irregular, unknown in advance, and often changing in structure.
XML (Extensible Markup Language) has emerged as a universal data exchange format for the Web. It is capable of representing data structure in text and has commonality among different types of data sources, in the sense that almost any data source can be converted to XML format. Although it is not classed as a data modelling technique such as those discussed above, it currently provides a welcome means for describing structure in Web-type data. It should be noted that XML is only a mark-up language and does not have as associated data model as such.

Like HTML (Hypertext Markup Language), XML is a subset of SGML (Standard Generalised Markup Language). The latter is a sophisticated tag language that has not

achieved widespread uptake due to its complexity and the complexity of the tools required. While HTML is used to describe pages to the Web by making use of tags that are interpretable by browser software, XML allows tags to be defined by users. Because of this approach, software cannot provide more interpretation to the structure, unless it is specifically written to do so. XML is therefore most useful in a community defining a set of common tags for its purpose, i.e. a type of meta-model. XML can however be augmented with an optional document type declaration (DTD). A DTD states what tags and attributes are used to describe content in an XML document, where each tag is allowed, and which tags can appear within other tags.

Limitations in XML's ability to represent the finer points of data structure, ability to recognise sub-types and constraints, as well as a lack in facilitating modularity and reuse, have prompted the World-Wide Web Consortium (W3C) Working Group to develop a new generation of schemas for XML. The DTD approach is often considered to be more of a grammar-based concept, ensuring the *well-formedness* of an XML document. In order to ensure the *validity* of an XML document, W3C have introduced XML-Schema. W3C's requirements [W3C Malhotra, 1999] suggest that this should include key semantic modelling concepts. The list includes structural, datatype, and so-called conformance requirements.

- *Structural requirements* : This includes mechanisms for constraining document structure and content, mechanisms to facilitate inheritance, an ability to "reference" the standard semantic understanding of a particular construct, mechanisms to specify application specific constraints and descriptions, mechanisms for addressing the evolution of schemata, and also mechanisms to enable the integration of structural schemas with primitive data types.
- *Datatype requirements* : Allow for the definition of primitive data types, define a type system that allows for import and export from different database systems, distinguish requirements concerning lexical data representation from those governing an underlying information set, and to allow the definition of user defined datatypes.
- *Conformance requirements* : These essentially include aspects that ensure the validity of XML components and their relationships. The XML Schema Language should therefore be capable of defining the relationship between schemas and

XML documents as well as playing a meta role by defining a useful XML schema for XML schemas.

In spite of being a new concept, the notion of XML schema deserves serious consideration in the data modelling techniques arena. It is similar to the notion of object-oriented modelling discussed above, in the sense that it also concerns design phase concepts.

XML as a means for building evolvability into structural specifications is considered in section 4.1.2.2.

Having introduced the primary processes and artefacts involved in designing and developing an information system, we next consider the problem that arguably presents the biggest obstacle to the successful endurance of an information system – evolution. This is discussed in terms of the impact on each of the SDLC phases.

CHAPTER 3 - THE PROBLEM OF EVOLUTION

For long-lived software projects, such as large information system applications, the operational and maintenance phase of the life-cycle is by far the longest. Such systems continuously undergo changes arising from both user requirements, as well as changes to the environment in which the system operates.

Researchers have specifically identified the field of *software evolution* as the area of study concerned with maintaining a system's structural and behavioural consistency after parts of a system have been changed. Evolutionary changes to a system can occur at various stages in its life-cycle for a number of reasons. These essentially correspond to the adaptive and perfective maintenance categories discussed in section 2.1.5. Reasons for their occurrence include the following:

- Optimal solutions are not always readily apparent. Better designs at both component and the greater architectural level may only be possible after implementation and actual experience of an operational system.
- User and organisational requirements change. Additional functionality must be integrated into the existing system.
- The underlying application domain which the system models undergoes change - requiring that the system follows suit. As Falkenberg et al. [Falkenberg et al, 1992] state, modern organisations must be flexible and adaptive in order to remain competitive in the global market place. Consumer needs are also becoming more demanding, and the need for information systems which can be easily adapted and evolved to the same extent as the information needs change is becoming crucial.

In particular, it is the unforeseen changes that occur quite frequently, and need to be respected in information system developments. These are typical of the last category.

In practice, development phases are repeated during the maintenance phase. The following sections therefore consider the phases of the SDLC as regards software evolution, with special focus on the subsequent issues. In particular, section 3.1.1 considers the requirements analysis phase. Section 3.1.2 considers specification, with emphasis on the challenge of building evolvability into conceptual models. Section 3.1.3 discusses the problem of design erosion as it affects software architectures.

Finally, the implementation issues are described in section 3.1.4 with emphasis on the schema evolution problem. This characterises the consistency and change propagation requirements that arise following an evolution. A brief consideration of the underlying problem of disparate implementation components as well as the need for a "supervisory" meta-model framework conclude the chapter.

3.1 Evolution and the SDLC

3.1.1 Requirements Analysis

As introduced in section 2.1.1, this phase aims to achieve a better understanding of users' needs as it proceeds. The phase itself is iterative, and concludes when a requirements definition document is produced where both functional and non-functional requirements are stipulated.

Changing user requirements on a new "green-fields" development is easily dealt with, particularly if later development phases have not yet been tackled: here, change propagation to design and implementation artefacts becomes a serious issue.

However, large established information systems are characterised by having long life-times spanning several years, if not decades. Functional changes to the underlying application domain are especially significant as they must be specified at the earlier stages of the life-cycle, and hence define the basis to which later design and implementation phases must adhere. Such changes usually arise due to changes in market, legislation, economy, and so forth. For instance, the production of new products or changes to the primary process of an organisation can alter the original application domain where new requirements may be as demanding as those that directed the initial construction.

Incorporating these changes into a requirements document is obviously necessary in order to ensure an accurate and consistent documentation artefact of the system. Most software engineering practices encourage that the inevitability of change should be recognised and anticipated when producing a requirements document. The document should be organised to accommodate easy editing and revision. [Sommerville 1992] mentions minimisation of external references and modular document sections as factors influencing changeability in documents. Electronic tool support is also

regarded as a far more effective and efficient vehicle for managing change control, as opposed to unwieldy paper-driven systems. However, these can still be limited to the foresight that exists at the time of creating the initial document, and may not necessarily accommodate those requirements changes that occur once a system has reached the operational and maintenance phases of its life-cycle. As studies by Lientz and Swanson [Lientz and Swanson, 1981] and Banker et al. [Banker et al, 1993] have shown, it is not unusual to encounter a whole class of problems that only show up once a system becomes long-lived, typically involves persistent data, and grows in complexity and diversity.

Of even greater concern is Sjoberg's [Sjoberg 1993] remark that most documentation is in fact notoriously poor and virtually always obsolete. The only reliable, up to date program information may be the source code itself or information that is automatically generated from source code. Even where documentation does exist, the incorporation of new requirements is challenging in order to preserve the integrity and consistency of the requirements definition. As mentioned in section 2.1.1, a requirements document that is amenable to a form of logical reasoning also bears advantages for the next stage which must deal with the more formally oriented system specification.

3.1.2 System Specification and Conceptual Modelling

This phase is characterised by the development of more formally-oriented models to represent the structural, behavioural and process or activity-related elements of an application. As this work is primarily concerned with the database aspect, focus is given to that which specifies the structure, behaviour, and relationships of objects in the application domain, i.e. the conceptual model. The structural framework generally represents the most static and stable view of the underlying Universe of Discourse (UoD). We begin by discussing the nature of the evolvability requirement for such structures.

3.1.2.1 Evolvable Conceptual Models : Challenges and Issues

Developing the conceptual model for an information system is a challenging task - unfortunately with few guidelines and decision criteria to assist engineers. Intuition and design experience are heavily relied upon. One of the most relevant issues

concerns the fact that a large number of correct solutions may be produced for a given Universe of Discourse, but as Verelst [Verelst 1997] notes, each final model possesses significantly different characteristics in terms of understandability, maintainability, redundancy, enforcement of business rules, stability, and so forth. Simsion and Shanks [Simsion and Shanks, 1993] provide empirical evidence to support this where fifty novice- and expert data modellers used the Entity-Relationship technique for representing a small-scale requirements set. The main conclusions were that each resulting model was both different and correct to an acceptable level, and that there was considerable variety amongst the models, both in the number of entities and relationships used as in the use of generalisation.

Researchers are challenged by providing guidelines to assist engineers in producing conceptual models that satisfy the *evolvability* quality for information systems. However, the first issue is to obtain a handle on the notion of evolvability at a conceptual level.

Work by Wedemeijer [Wedemeijer 2000] and Verelst [Verelst 1997] tackle this, both from theoretical as well as more empirically-oriented approaches.

Wedemeijer [Wedemeijer 2000] indicates that a conceptual design is required to be stable enough to support a long-term systems lifetime, and be flexible enough to meet future information demands. Flexibility essentially concerns the adaptability and responsiveness of a model to future changes. Greater flexibility results in a smaller impact of change. Stability is similar, but where flexibility refers to a future capacity for change, stability refers to the history of the model in the sense that it is achieved if required changes have been accommodated : stability is proof that flexibility has been delivered.

Flexibility

We begin by considering flexibility as the more traditional measure of a conceptual model's quality to accommodate change. According to Wedemeijer [Wedemeijer 1999], three main design strategies exist that are widely accepted as delivering flexibility. These include:

- *Active flexibility or adaptability*: This aims to improve the design by arranging the constructs of the model in such a way that it is easy to modify. Normalisation, modularization, incremental design and the use of component libraries (if applied to reduce the time for response to a change) are based on this strategy. These are however inherently plagued by difficulties. Firstly, the arrangements of constructs are biased in that it assumes that future changes will be of the same type that the design was originally geared to handle. This issue relates to Verelst's [Verelst 1998] concern regarding variability in horizontal abstractions and involves the choice among concepts on which to *base* the structure of the model on. He notes that it can be possible to treat all concepts as equivalent, but that certain primary dimensions are usually (unconsciously) chosen to determine the structure of the model.
- *Passive flexibility*: This aims to decrease the need for future change in the model by incorporating more requirements into the design than those originating from the current Universe of Discourse. Wedemeijer [Wedemeijer 1999] lists reuse of "proven" designs, Business Data Modelling and the use of component libraries representing "good solutions" as examples. Once again, fundamental problems include how far ahead future requirements should be anticipated, which requirements are relevant, and which are beyond consideration. Creating models that are "over-flexible" can also lead to weaker constraint-specifications and a situation where "anything is possible".
- *Flexibility by abstraction*: Such strategies aim to put less information into a conceptual model, thereby making it more abstract. Designers are however challenged in having to decide on the best level of abstraction. Verelst [Verelst 1998] notes that models can become difficult to understand as abstraction increases. Furthermore, he cautions that abstractions can also define a certain "evolution path" where changes can be easily made, but changes outside of this scope can be awkward and inelegant. The transformation and relationship of abstract conceptual models to workable external and internal schemas can also be problematic.

In essence, Wedemeijer's [Wedemeijer 1999] challenge to the claims of flexibility made by these, and other such approaches, centres on the following: '*why they should*

enhance flexibility is often explained, sometimes demonstrated, but rarely proven by actual business cases'. In summary, considering flexibility alone as a measure for accommodating change is unsuitable since:

- Flexibility can only be established "on the fly". A potential for change cannot become apparent on a new model, but only when a structural change occurs in the Universe of Discourse.
- There is no distinction between structural changes that ought to be accommodated by the flexibility in the design and those beyond the desired flexibility, and
- There is no way to verify that a given design has sufficient flexibility, or to discover that more is needed.

Stability

Given the emphasis on observing conceptual models in operational business environments and the issue of change over their operational life-times, Wedemeijer [Wedemeijer 1999,2000] suggests the study of stability. However, there is no generally accepted and unambiguous definition of the concept of stability. Wedemeijer [Wedemeijer 1999] suggests that a change in a conceptual model is a stable change if it is absolutely necessary to accommodate a change in the structure of the underlying Universe of Discourse. Any other change is deemed unstable. In an operational environment, enterprises try to keep the impact of change as small as possible and will naturally restrict the freedom of choice when adapting a conceptual model. The difficulty arises in ensuring that the adapted model is a good model of the changed Universe of Discourse, while still being as "close" as possible to the former conceptual model. This relates to demands for compatibility and extensibility in models. However, an underlying problem still remains, i.e. determining if changes are indeed stable: few guidelines and metrics are available to assist designers. Both Wedemeijer [Wedemeijer 1999,2000] and Verelst [Verelst 1997,1998] emphasise the need for empirical studies on actual business cases. Here, the relationship between changes in the Universe of Discourse and the operational conceptual model can be better understood. Unfortunately, such studies can be hindered, as changes are difficult to observe in real business environments. Studies that have been conducted

have either described symptoms of the stability problem rather than its essence, or have used limited or over-simplified taxonomies for detecting change.

Lastly, it must be stressed that although factors such as the degree of encapsulation, information hiding, or abstraction in models can bolster the stability and also future evolvability of a model, the problem remains that multiple correct conceptual models can be built with similar levels of each factor, but yet have different evolvability characteristics. Once again, empirical studies are required to verify theoretical claims.

3.1.2.2 Behavioural Evolution

While recognition of evolution at the structural conceptual level is necessary, so too is behavioural evolution. As Saake et al. [Saake et al, 2000] indicate, the rules or axioms describing the allowed dynamic behaviour of entities may indeed change during the existence of those entities. He also comments that neither well-known approaches to conceptually describing information systems, such as UML, nor formal specification approaches, provide adequate support for dealing with changing requirements. Methods are sought whereby changes in the behavioural aspects of a conceptual specification can be accommodated.

Furthermore, methods are also required whereby changes to the behavioural specification are consistent with the structural conceptual specification. Some form of meta-modelling would appear relevant as a means to control this, but the nature of its application requires further study such that a coherent framework is provided for designers.

Design components and architectures are prone to the evolvability shortcomings of specifications, as they are essentially a more refined interpretation of conceptual artefacts. However, additional issues also arise. These are now discussed below.

3.1.3 Design

The early stages of the design phase for an information system are typically characterised by the creation of a design architecture where the different components are specified in terms of their relationships with one another. These components include software entities that handle data storage or computation for some aspect of

the information system. As indicated in section 2.1.3, this decomposition may comprise functional- or object-oriented elements. The aim is to realise a means to demonstrate that an eventual implementation of the various components will indeed satisfy stakeholders' requirements.

3.1.3.1 Design Components and Software Architectures

To facilitate maintenance, designs should be readily adaptable. This suggests that the components be loosely-coupled. Furthermore, components should be self-contained where usage or dependencies on externally defined components is minimised. As [Sommerville 1992] notes, this is somewhat contradictory to the practice of component reuse. Hence, a trade-off exists between the advantages of reusing components and the loss of adaptability that this entails. The reusability aspect is also relevant to the evolvability of the component model. This suggests that the adaptable or evolvable parts of components be differentiated.

Object-oriented systems are amenable to adaptation and reuse in the sense that the adaptation mechanism does not rely on modifying the component, but rather on creating a new component whose attributes and operations are inherited from the original component. The original component and its dependants remain unaffected. However, for long lifetime systems, object-oriented systems require careful management in that their inheritance network can become increasingly complex as changes are made. Duplication of functionality can also result leading to redundancy issues.

This addresses maintenance at a relatively low design level. Developers also require a better handle on the gross organisation of the system. The field of software architectures emerged as a natural evolution of design abstractions to address this issue.

Software architectures were introduced in Chapter 2 as a valuable design methodology for information systems. Although structural, behavioural and control related aspects are described by different models, a software architecture must be aware of each, together with the mappings that exist between them in order to present a unified methodology for understanding the overall system. To further speed

development and facilitate reuse at an architectural level, different architectural styles emerged – design patterns, for example, have emerged as a means for the reuse of a solution for a specific design problem. They are usually considered in close connection with object-orientation and describe, in abstract terms, how a general arrangement of design elements (classes and objects) can solve a problem. Domain-specific architectures, in particular, are popular for business information systems, where business processes and objects are organised and related to guide development of a new system. The sd&m architecture overviewed in [Hoffman 1996] is an example. As the design proceeds, lower level and more refined views of certain architectural aspects (e.g. components and connectors) are available.

Unfortunately, fundamental problems remain in terms of design components and architectural structures. This concerns a lack in the capability to accommodate (isolate the effects of) changes and facilitate (assist and aid mechanisms or processes effecting) change.

3.1.3.2 Design Erosion

Although the likes of object-orientation and software architectures greatly benefit the construction of software, they do not adequately address the accommodation of changes during later operational stages. As Van Gurp and Bosch [Van Gurp and Bosch, 2001] note, software designs do indeed erode over time; to the point that redesigning from scratch becomes a viable alternative compared to prolonging the life of an existing design. Unforeseen requirements changes can invalidate design decisions that were once optimal. They suggest that design erosion is caused by a number of problems associated with the way in which software is commonly developed. These include:

- *Lack of traceability of design decisions*: Notations used to create software can lack expressiveness that is needed to express concepts used during design. This results in difficulties when attempting to track and reconstruct design decisions from the system.
- *Increasing maintenance cost*: Over time, the complexity of the system can increase. This leads designers to consider sub-optimal design decisions, either

because they do not understand the architecture, or because a more optimal decision would demand too much time and effort.

- *Accumulation of design decisions*: When a design decision needs to be revised, so do other design decisions require reconsideration, possibly leading to developers having to work with a system that is no longer optimal for requirements it must now meet.
- *Iterative methods*: A "proper" design is expected to accommodate future change requests. This, however, conflicts with the iterative nature of rapid prototyping development methods. These progressively incorporate new requirements but also have shortcomings. Sommerville [Sommerville 1992] notes that for large, long lifetime systems in particular, prototypes should be re-implemented anyway. This is due to characteristics such as performance, security, robustness and reliability usually being ignored during prototyping. Furthermore, as prototypes are changed to incorporate new requirements, it is likely that these changes are made in an uncontrolled way, resulting in the prototype code acting as the only design specification. This is inadequate for long-term maintenance. Lastly, prototypes tend to suffer from the "accumulation of design decisions" problem. Here, changes made during prototype development could have easily degraded the system structure, so that subsequent maintenance requirements become progressively more difficult to make.

Good design methods, such as separation of concerns which can isolate the effect of changes, using sound design and architectural patterns to guide the design process, and in general, designing for change, are all noteworthy for delivering better designs but do not address the fundamental problems that cause design erosion. Van Gurp and Bosch [Van Gurp and Bosch, 2001] state that they only contribute by delaying the moment that a system needs to be retired.

By using an experimental system, he was able to conclude that causes for design erosion problems did indeed range from accumulation of multiple design decisions (i.e. certain design decisions were taken because of earlier design decisions, even if they were the wrong decisions), to limitations of the object-oriented paradigm (e.g. inheritance constraining flexibility at runtime, encapsulation forcing objects to only interact via method parameters). Even optimal design strategies (i.e. no compromises

concerning cost or effort) proved to be insufficient for accommodating change in later evolution cycles.

Andrade and Fiadeiro [Andrade and Fiadeiro, 2001] have stated similar findings, where object-oriented techniques do indeed simplify the combination of components in a way that reflects interactions in the underlying application domain. However, changes on the implemented systems resulting from accommodation of new business rules cannot be performed in such a modular way. He indicates that this is due to interactions being "hard-wired" into code that implements the participating objects, thereby making it difficult to change or introduce new interactions without having to change the implementation of the objects as well. This in turn may have a ripple effect throughout the implementation of the system, compromising the architectural integrity. [Andrade and Fiadeiro, 2001] also indicates that while lower-level design strategies, such as those offered through design patterns, can deliver more flexible solutions, they are too low level to support an evolution process that takes place at higher levels of abstraction where business strategies and rules can be redefined.

As the system enters its implementation and operational phase, the growth of persistent objects (both program and data) in particular, becomes an overhead that must be carefully managed by any evolution system.

3.1.4 Implementation and Operation

The developer is now required to realise the design artefacts from the previous phase in terms of program and code destined for a particular execution platform. Executables are delivered and users may begin to test and evaluate the system in liaison with the developer. As introduced in section 2.1.4, a cycle of validation and verification commences, which typically includes user requirements changes that need to be addressed. This problem is exacerbated once the system becomes operational. Persistent stores become populated with data, interruptions to working systems to accommodate maintenance are seldom acceptable, and more applications are developed that are associated around the database schema defining the persistent store. Future evolution requirements may also require that the system be able to integrate (horizontally) with other systems – the world-wide web all but demands this

from information system infrastructures. The effort involved in maintaining systems at this stage is in itself extremely significant : as the scale and complexity of systems increase, so too does the requirement for effective and usable tool support.

The following areas are of interest to the evolution problem, particularly at the operational stage:

- Schema and database evolution
- Application (behavioural) evolution
- The need to accommodate and ensure *overall* consistency of all information system components when faced with change.

The field of system re-engineering is also relevant as it deals with legacy systems where modification and evolution to meet new and constantly changing requirements is resisted to such an extent that the system must be rewritten, or be completely or partially restructured. This research is limited to approaches that avoid resorting to re-engineering and is more focused on methodologies that realise developments able to accommodate and facilitate evolution. Unfortunately, the extent and reliance on operational legacy systems cannot be underestimated, often necessitating substantial re-engineering efforts.

We now discuss the areas that aim to address evolution of operational systems without unduly comprising the original development. This includes research that deals with the problem reactively and well as proactively. At this juncture, we focus on the issues and considerations regarding operational stage evolution in order to provide an indication of the types of solutions required.

3.1.4.1 Schema and Database Evolution

Schema evolution is essentially concerned with modifications of the database schema in such a way that conceptual consistency with respect to the underlying domain is maintained. A system supporting evolution would essentially be one where the database schema can evolve without the loss of any information. Following a schema change, database objects must also be consistent with the modified schema.

In particular, the following challenges need to be addressed:

Semantic Integrity

In order to ensure that schema evolution mechanisms are indeed consistency preserving and correct with respect to the underlying domain, an appeal must be made to design techniques applicable at earlier stages of the development life-cycle. This would typically included a need to relate conceptual schema designs, where one would aim to capture domain semantics formally, to internal database schemas on particular platforms.

Architectural Issues

Evolution may be achieved either completely, incrementally, by versioning, or by view-related approaches.
Traditional techniques generally perform a complete evolution involving an entire recompilation of the schema, but with applications having to be suspended. Incremental mechanisms are generally facilitated via primitives. Each primitive is atomic by nature thereby promoting consistency and the potential for reversible modifications to the schema. Impact on running applications is less severe as modifications tend to be done in a more on-line mode. Modifications using this approach tend to be limited to simple schema changes. Tool support, in particular, is sought for managing more compound-oriented changes on types, such as the merging of object-oriented classes.

Following a schema change, database objects must be consistent with the modified schema. Database availability and application compatibility are of particular concern. Approaches can either involve adaptation of the actual database objects, or rely on some mechanism to support "emulation" if objects conforming to one schema version have to be seen as objects of a different schema version.
Adapting database objects involves the use of either immediate or deferred data transformations. Immediate transformations result in the entire database being in a state consistent with the new schema, but database availability is compromised.

Deferred techniques solve this by converting data objects on an as needed basis, but also result in a data access overhead and requirement for tracking and managing the history of updates.

An attractive alternative to data conversion includes schema versioning and view-based approaches – particularly for their intended lack of impact on database availability and application compatibility. Schema versioning techniques, for instance, aim to allow access to all data via both retrospective and prospective user definable version interfaces and, if possible, be extended to facilitate the update of data through historical schemata. However, while existing applications should experience minimal disruption, there is considerable overhead in managing multiple versions that access a single database. Following a similar approach, view support techniques are also attractive but are generally limited in their support for allowing updates or the addition of new data, as might be accomplished by adding new attributes to classes in an object-oriented database.

Recent solutions to tackle the schema evolution issues presented above are considered in section 4.3.1.

3.1.4.2 Application (Behavioural) Evolution

Although the notions of compatibility via views or schema versioning approaches are attractive for the likes of legacy applications, implemented applications must themselves evolve.

In particular the following requirements must be addressed:

- *Semantic integrity and consistency of the change process*: Any change should be guided and constrained by the underlying specification.

  Furthermore, changes to the database schema of the system will require modification to the application. Traditional approaches are generally capable of compiler-oriented warnings. Although useful, they are mostly limited to syntactical checks. Mechanisms are required that are more semantically-oriented, thereby facilitating a basis for tool support and a more automated means of ensuring that semantic integrity is preserved.

- *Dynamic evolution*: Applications, whose behaviour is dynamically determined by changes in the underlying domain, require advanced program restructuring techniques. An example would be a generic data structure browser where, on receiving a specification of the data structure, must automatically generate a program to browse over it. Application areas such as geographical information systems, CAD/CAM systems and multimedia systems, are also prone to requiring the facilitation of some sort of dynamic behavioural evolution.
- *Better integration with persistent store technology*: The impedance mismatch problem concerns the fact that database query languages are inherently declarative and oriented to set-level processing, whereas programming languages are mostly procedural and oriented to record-at-a-time processing. Besides the overhead in the programmer having to manage the interface between the two, an overhead is also created for any required evolution management on the system. In particular, the propagation of change to ensure consistency and system integrity becomes complicated. The need for some sort of wholistic view is required, and is considered next in the broader context of all system components, including user-interfaces and operating system interfaces.

3.1.4.3 A Wholistic View : Issues and Considerations

To re-emphasise, and as Sjoberg [Sjoberg 1993] indicates, the issue of ensuring consistency in the change propagation process for an information system is complicated by the fact that they are generally centred around a database. Changes to database schemata (schema evolution) may in turn have serious impacts on other parts of the schema, on extensional data, and on application programs (including interfaces for data entry, queries, report generation, etc.). Dealing with evolution separately among these components has been identified as impractical, inefficient and susceptible to corrupting the integrity of the system as a whole. This is also suggested in the 1998 Asimolar Report on Database Research [Asimolar 1998] where, from the database point-of-view, it is noted that code is not a first class object and co-equal to data in current database systems. The report also states that database systems need to be more application aware to facilitate the likes of large-scale system integration.

Better techniques are required for managing descriptions of application interfaces along with higher-level model-driven tools leveraging these to help integrate, evolve, migrate and replace application systems. Research work has also identified the following:

- Some form of meta-modelling and meta-programming (possibly in combination) is required. The former helps to unify the different software components that ultimately comprise an implementation, in the sense that models are defined and constrained by meta-models, while meta-meta models can attempt to provide a global integration for all the meta-models in the software development scene. How this might be realised is an area receiving much research interest from both academic and vendor quarters. We consider this in section 4.3.4.
- Disharmonies and incoherence in current implementation technologies is a fundamental contributor to evolution problems. Applications rely on disparate mechanisms including operating systems, communications systems, database systems, user interface systems, command languages, editors, file systems, query languages, etc. [Atkinson and Morrison, 1995] identifies Persistent Application Systems (PAS's) as long-lived, concurrently accessed, and potentially consisting of large bodies of data and programs. They typically outlive their individual components and implementation technologies. The aim, therefore, is to realise a coherent, wholistic design approach that eliminates these disharmonies and unnecessary sources of complexity, such as the impedance-mismatch problem. Atkinson's work on orthogonal persistence ([Atkinson and Morrison, 1995]) represents the main thrust of this research direction, and while attractive, also requires considerable effort in resolving numerous issues such as integration of types, data models, binding mechanisms and concurrency control between programming languages and databases. Technology to support such a system is also an issue. Orthogonal Persistence as a solution to the evolution problem is discussed in further detail in section 4.3.3.

Chapter 4 considers different solutions proposed in the literature towards accommodating and facilitating evolution. The format is similar to this and previous chapters, with solutions presented in terms of their relevance at the different stages of the SDLC.

CHAPTER 4 – DEALING WITH THE EVOLUTION PROBLEM

This chapter considers specific solutions to the problem of evolution through the information system development process. Issues and difficulties regarding evolution were introduced in Chapter 3 by considering the different phases of the SDLC and the effect of change at each stage. Although the problem is far from solved, many contributions have been forthcoming, and collectively suggest criteria that information system developments must consider if they wish to successfully endure the inevitable change requirements that arise over an application's lifetime. To begin with, this chapter follows a similar approach to previous ones by considering solutions at specific stages of the SDLC. However, the need for a more pervasive framework governing all stages emerges as a distinct requirement in order to manage the promulgation of change from requirements analysis stages, through to implementation artefacts.

No one particular method or methods were considered in detail for the requirements analysis phase. Recommendations and considerations are rather discussed in a synopsis presented in Chapter 5.

We begin with proposals presented in the literature towards improving stability of specifications. This is discussed in section 4.1.1. The notion of explicitly specifying change is then considered in section 4.1.2 for both behavioural and structural evolution. This is followed in section 4.1.2 by a discussion regarding approaches enabling the accommodation of change in design architectures. Section 4.3 considers the schema evolution problem and its associated concerns, while section 4.3.2 briefly discusses reflection as a technique for effecting dynamic behavioural evolution. Section 4.3.3 discusses orthogonal persistence and the PJama project as a solution for improving and simplifying disparities in implementation technology, while section 4.3.4 describes a meta-modelling implementation framework : the OMG's MDA approach, in particular, is seen as beneficial towards providing a more model-based approach to information system development.

4.1 Evolution-friendly specification

As was indicated previously, current conceptual modelling techniques lack guidelines to assist designers in assessing the evolvability capabilities of a given model. This also applied to models where strategies such as encapsulation and abstraction were used. Furthermore, authors like Sjoberg [Sjoberg 1993] have indicated that traditional notions of stability in teaching and practice, data modelling, data schema construction, and so forth, must be breached in order that change be sufficiently accommodated.

Two major aspects towards improving current practice in conceptual specification of information system structure and behaviour prevail in current literature. The first recognises that well-designed conceptual models will remain stable over time, and explores how designers might measure this stability over the operational life-time of the model in order to deliver higher quality designs that stand the test of time. The second aspect recognises that current modelling and specification technology does not allow designers to model or specify changes that might occur during the operational lifetime of a system in a flexible way. The degree to which these changes are successfully accommodated may then be appropriately measured in terms of the resulting stability.

4.1.1 Stability Characteristics

A conceptual model is generally regarded as the best means of perceiving a Universe of Discourse (UoD), not only at design time, but also as they both evolve over time. According to Wedemeijer [Wedemeijer 2000], a model suited to evolution would be one delivering stability, in the sense that any required changes have been accommodated and that flexibility has been delivered. The issue his work addresses in particular, is that current literature rarely addresses how such stability should be observed and measured in the operational business environment with evolving information needs and data structures. In order to attain a better understanding of the actual mechanisms involved in exploiting flexibility as a potential for change, Wedemeijer [Wedemeijer 2000] presents hypotheses and associated metrics on how conceptual schema stability ought to be expressed in operational environments. These are briefly discussed below:

- *Justified change:* A change in the conceptual structure is only justified if it is a change in the UoD's information structure that is causing it. The metric for this is the ratio of single conceptual model changes that can be associated with an appropriate change driver from the underlying UoD, to the total number of conceptual model changes that had to occur, regardless of whether they were associated with an apparent UoD change or not. Ideally, the ratio is equal to 1.
- *Proportional change:* Every change in the conceptual model should be proportional to the change in the UoD that caused it. A small UoD change leading to a large change in the conceptual model would imply an unstable model. Measuring the size or severity of such a change can be somewhat subjective. As a guideline, Wedemeijer [Wedemeijer 2000] suggests a comparison between the number of paragraphs explaining the change in the UoD, to the number of affected constructs in the conceptual model.
- *Proportional rate of change:* This essentially suggests that the rate of change in the conceptual model should be proportional to the rate of change in the UoD. Once again, there is a risk in precisely quantifying new user requirements and the lifetime of consecutive sets of user requirements against the number of conceptual model changes and the lifetime of consecutive conceptual model versions.
- *Compatibility:* A new conceptual model is considered to be compatible with the old one if no data instances in any construct of the old model needs to be altered or discarded in order to suit the new model. This effectively eases any subsequent schema evolution overheads. Wedemeijer [Wedemeijer 2000] measures the extent of this by considering the size of the "external" view on the old conceptual model describing the affected data, relative to the size of the original conceptual view. Changes in the level of abstraction are regarded as particularly difficult to accommodate, as semantic discrepancies between versions require consideration.
- *Extensibility:* New requirements should be catered for by extension or addition of new conceptual constructs, as opposed to modification of existing constructs. The latter measures the extent of non-extensibility of the model. As this suggests, changes by extension should leave old data instances fully compatible with the new schema.
- *Complexity hampers change:* A typical measure of the complexity of a conceptual model would be the number of components in the model, the number of ways in

which they are interrelated, and how these interrelationships may change over time. Increases in complexity can be expected to lead to greater difficulties in accommodating change. A more complex conceptual structure should therefore change less frequently, indicating a favourable measure for stability.

- *Abstraction reduces the need for change:* Related to the notion of complexity is the concept of abstraction. Designs that are more abstract are generally considered to be more stable due to a lower number of constructs that must be adapted to new requirements. The level of abstraction in the model, compared to the number of construct changes, can therefore provide some measure of stability. However, measuring the degree of abstraction in a model is a debatable issue.
- *Susceptibility to change:* This recognises that some types of constructs in the model are more susceptible to change. Entities and relationships are presumed to have best stability, then attributes and relationship cardinalities, while integrity constraints and business rules are most volatile. Observing the number of changing constructs and their type could thus assist in measuring stability.
- *Preservation of entity identity:* The means for identifying entities, such as candidate keys in relational data model theory, should not be changed. Such change is only acceptable when the entity itself is observed to change.
- Further hypotheses are listed, viz. change is local, change is restricted to a single module, and modules are stable. These generally recognise the notions of localisation, cohesiveness, and loose-coupledness as bettering stability.

These hypotheses and metrics are geared at assessing the stability of the constructs of a conceptual model when faced with changes from the underlying UoD. Although relevant in parts, they do not explicitly relate to behavioural modelling concerns as such. However, we can also appeal to the likes of Balzer's [Balzer 1986] work on properties of good specifications which was introduced in section 2.1.2. Localised and loosely-coupled specifications, for instance, should help in bolstering the stability of a specification. The use of "demon" capabilities, that act as independent agents for dealing with environmental changes, are claimed as being useful for absorbing change impacts that would otherwise cause overheads in maintaining interactions and relationships between system components – a high-level architecturally-related concern in particular.

Designers, however, require solutions and frameworks where changing requirements are accommodated and conceptual stability maintained. These are considered in the next section.

4.1.2 Specifying change

Following the requirements definition phase, the designer typically begins to consider means to specify the system more formally. Numerous techniques exist to accomplish this and primarily aim to establish a semantic model for representing structural and behavioural aspects of the information system. As was discussed in section 3.1.2, the phase is complicated by the fact that multiple models can be built – all being correct as far as describing the application domain is concerned, but ultimately delivering varying degrees of evolvability. Even recent technologies such as the Unified Modelling Language (UML) are noted as not providing adequate support for dealing with changing requirements. It is especially the case of unforeseen changes that must be considered and catered for in a flexible way during the lifetime of a system.

Work by Saake et al. [Saake et al, 2000] explores this with the aim of supporting continuous engineering of information systems at the conceptual level. His work takes cognisance of the fact that the behavioural aspect is the most volatile, and therefore appropriate to business rules which change over time. Here, the object-oriented model is considered in terms of accommodating an evolving behavioural specification and is now described in further detail below.

Building evolvability into structural specifications is then considered in section 4.1.2.2.

4.1.2.1 Modelling change in behavioural specifications

Traditional approaches require the behaviour of objects to be completely fixed at specification time, in the sense that dynamic behaviour is not modelled. Saake et al's [Saake et al, 2000] proposed solution stems from the premise that information systems consist of large numbers of long-lived objects, and that over time, conceptual

level requirements such as business rules and laws evolve. The associated system evolution therefore leads to changes in not only objects' states, but also the rules (or axioms) which describe the allowed dynamic behaviour of objects. Changing requirements during runtime therefore leads to new axioms being added, or existing ones being removed or changed. This may seem similar to implementation-level approaches, such as those used in SQL database management systems where insertion, modification and deletion of SQL functional units such as constraints, triggers and stored procedures, are supported during runtime of an application. However, to re-emphasise, the issue here is to support continuous engineering at a conceptual level, and to strive toward a more formal basis for specifying change. This allows later development stages to be better controlled, and formal reasoning at the conceptual level is in turn facilitated.

Saake et al. [Saake et al, 2000] begin by using TROLL : a formal specification language-based technique with clear semantic underpinnings. More popular object-oriented modelling approaches, such as OMT and UML, are cited as lacking clarity and being too restrictive for this purpose. TROLL is able to provide a framework for formally specifying structural as well as behavioural aspects of information systems. An extension is proposed that can cope with representation of dynamically changeable behaviour. Work by Balko [Balko 2000] also advocates such extensions to TROLL, particularly for the specification of an industrial production environment where the workflow specification for a group of machines is subject to dynamic change.

According to [Saake et al, 2000], the extended framework is based on the following concepts:

- During design, a separation of the rigid and the evolving part of application objects has to be performed.
- A rigid specification level exists which fixes the signature of application objects as well as basic functions.
- The evolution level of the specification manipulates specification fragments whose vocabulary is identical to that of the base level.

- Critical functions should be part of the base level. They are then safe from undesired modifications during evolution, and their properties can be formally verified through conventional approaches.

To illustrate this, consider the example shown in Figure 4.1 below (adapted from [Saake et al, 2000]). This shows the signature and behavioural specification for a system which must cater for basic document management.

```
object class Documents
 identification DocID: (DocNo)
 template
  attributes
   DocNo: int,
   DocType: {offer,contract},
   Valid: date,
   Content: text;
  events
   birth create(DocNo:int,Content:text),
   revise(NewContent:text),
   death resolve;
  ... //behaviour specification starts here
  rigid axioms
    create(D,C)
        changing DocType = offer,
                 DocNo = D,
                 Content = C,
                 Valid = now + 30
        calling  DocManager.addDocToOffers(self);
    revise(C)
        enabled  DocType = offer and Valid >= now,
        changing Content = C,
                 Valid = now + 30,
   ...
```

Figure 4.1 – Signature and Behaviour specification for document management system

Figure 4.1 indicates that documents can be uniquely identified by a *DocId* specified by attribute *DocNo*. Attributes are also listed, as well as events (actions) which can change the values of attributes or cause the occurrence of other events in other objects. The *create* event, for instance, creates new objects and sets the initial state of the object. The *revise* event allows the changing of the contents of the document. The (fixed) behaviour specification of events declared in the signature part is also shown. Here, its effects on attributes (*changing*) , its enabling condition (*enabled*), and its communication effects (*calling*) are specified.

Saake et al. [Saake et al, 2000] also indicate that linear temporal logic can be used at a semantical level to support reasoning concerning single objects. For example, the temporal logic formula

$$\forall C(\text{always}(\text{occurs}(\text{revise}(C)) \Rightarrow \text{next}(\text{Content}(C)))$$

states that it is always the case that if the revise event occurs with parameter C in a given system state, then in the next state the attribute *Content* has the value C.

As mentioned earlier, an evolution level is required to manage the specification of evolving object behaviour. This allows a separation between rigid axioms and evolving axioms. The latter represent the evolving behaviour part, allowing the behaviour of an object to be changed dynamically. Here, axioms are added or removed during runtime.

In order to deal with this at a specification language level, Saake et al. [Saake et al, 2000] introduce a special attribute, called *axiom attribute*, to store the currently valid set of evolving axioms. Events, called *mutators*, mutate the object's specification by changing the axiom attribute. These effectively change the behavioural description of the object at a meta level. Figure 4.2 below provides an indication of the constructs used in the extended specification language.

```
object  class Documents
 identification DocID: (DocNo)
 template
  attributes
  ...
  events
  ...
  rigid axioms
  ...
  axiom attributes
       Rules initialised{}
  mutators
      add_rule(Rule:spec)
       remove_rule(Rule:spec)
  dynamic specification
      add_rule(Rule)
          changing Axioms = Axioms + {Rule}
      remove_rule(Rule)
          changing Axioms = Axioms - {Rule}
end object class
```

Figure 4.2 – Extended specification to accommodate evolving object behaviour

This framework is constructed such that the same language constructs are used for manipulating the base and meta levels. For instance, mutator events can also be guarded by defining enabling conditions. Figure 4.3 provides an example of using the mutator *add_rule* to restrict the enabling of a resolve event such that "contract" type documents are never resolved.

```
add_rule (resolve
              enabled Doctype ≠ contract)
```

Figure 4.3 – Using the add_rule mutator

A fundamental issue with this approach lies in determining which part of the object behaviour is specified in terms of rigid axioms, and which as evolving axioms. Resorting to a behavioural specification comprised only of evolving axioms creates the problem that everything is possible. This hinders the ability to prove properties about the objects. Another inherent limitation concerns how far evolving behaviour

can be modelled in advance. A further issue is that a corresponding logic for interpreting evolving specifications is required: Saake et al. [Saake et al, 2000] propose an extension of the linear temporal logic introduced earlier called *Dynamic Object Specification Logic*. This provides so-called mutation event symbols and mutation attribute symbols to help model the semantics of mutations. The approach, however, requires consideration where the *always* operator is involved, as this influences the complete future of an object. The "state" of the specification therefore becomes relevant in order to determine the longevity of *always*-type axioms. This is noted as needing further research. Detail concerning the formalisation of this logic appears in [Conrad et al, 1998].

Lastly, this technique does scale to allow mutators to specify behavioural evolution at the class level. The granularity of changes can therefore be controlled to allow for exceptions to be dealt with at the object level.

4.1.2.2 Building evolvability into structural conceptual specifications

Having considered how an adaptive information system might be specified as far as accommodating behavioural change is concerned, we can also explore the equivalent notion for structural or schema-related specifications. This would aim to bolster the stability characteristics described earlier in section 4.1.1.

However, information systems typically centre around a database schema where changes inevitably lead to issues which must be addressed by schema evolution research, viz. maintaining semantic and structural consistency as well as propagation of changes to database instances and associated applications. These can result in a considerable workload, making the idea of enhancing the adaptability and robustness of a schema design worth pursuing. The problem would then be dealt with proactively as opposed to reactively.

This is explored as part of the EVOLVE project [Liu 1998] where adaptive specification techniques for object-oriented software evolution are considered. The techniques essentially involve the use of style rules, not only to verify desired properties of a schema design, but, if the schema is found not suitable, to also use these style rules as baselines to transform the schema into a better style while still

preserving semantics. Although the style rules are not generic (they focus on object-oriented schemas in particular), they do advocate some of the notions related to achieving stability. These were introduced in section 4.1.1. *Information localisation*, for instance, is stated as a rule for enhancing extensibility in an object-oriented class hierarchy through abstraction of common components. Liu [Liu 1998] promotes this by encouraging inheritance along specialisation hierarchies.
In general, the approach is limited in that it still tries to anticipate future requirements changes, and is therefore constrained by the foresight that exists at design time. However, application areas that are inherently evolutionary would seem promising.

In particular, the recent explosion of Web- and multimedia-based data has necessitated studies relating to information sources that are characterised by semi-structuredness and continuity (in the sense of persistent application systems introduced in section 3.1.4), as opposed to conventional DBMS technology which assumes formatted data and rigid database schema structures. Certain application areas, such as the modelling of biological data, are also evolutionary by nature. For example, characteristics of certain organisms may change over time, posing difficulties for conventional modelling structures. In general, the structure of this "non-traditional" DBMS data is sometimes irregular, unknown in advance, and often subject to change without notice. Solutions to this would appear to hold promise for the problem of accommodating evolution due to changing requirements. Proposals are discussed in the next section within the context of modern markup languages. These are receiving increasing attention as formalisms for data and knowledge modelling. However, their direct role in conceptual specification (as pertains to semantic modelling) is contentious, requiring careful consideration as was hinted at in section 2.2.2 where the XML markup language was introduced as a means for describing data on the World-Wide Web in particular.

Markup languages – A solution to modelling changing structure in Information Systems?

Modern markup languages, such as SGML and XML, are generic in the sense that they serve to specify structure as opposed to layout of documents and data items.

They do not impose any predefined structure, nor predefined names for the structural elements occurring in data items. As indicated in [Bry and Eisinger, 2001], it is possible to faithfully model the structure of data items needed in applications and to name the structural elements of a chosen structure in a way that is natural in the application context. The example in Figure 4.4 depicts an address book entry in XML-type format.

```
<person>
        <first_name> Harry </first_name>
        <last_name> Smith </last_name>
        <physical_address>
                <street_number> 4 </street_number>
                <street> Elm </street>
                <suburb> Oakwood </suburb>
                <city> Johannesburg </city>
        </physical_address>
        <telephone_number> 011-789-0005 </telephone_number>
</person>
```

Figure 4.4 – Address book entry in XML format

[Bry and Eisinger, 2001] also indicate that as the XML document stands in Figure 4, it is the use of application relevant names for structural elements that is at the origin of the expression of structure-conveying data. Data items structured in such a manner are not necessarily accompanied by a schema which might act as a specification of the structure of the underlying data items. This "absence" of a predefined schema structure is what has made XML attractive for modelling web content and other application areas such as modern biology. The following factors help characterise such areas and also suggest why markup languages may be useful:

- They are subject to general structural constraints, such as the biological building laws that describe relationships between biological entities, as well as the exceptions to those laws: an area not well catered for by traditional modelling formalisms.
- The underlying data items can be based on a multitude of data schemes as there is no generally accepted data model or ontology. The irregularities in structure are indeed another form of exception and suggest a case for modern markup languages.
- Data items in these applications are often enriched with texts – modern markup languages were designed for text.

However, as indicated earlier, XML documents generally rely on tag-naming and on the software interpreting the documents for revelation regarding semantic relationships in the application domain.

Therefore, in order to describe data in the same sense as more established formalisms, like UML for instance, the W3C developed the notion of XML-Schema introduced in section 2.2.2. Furthermore, recent research has investigated mappings from traditional semantic modelling techniques, such as ORM, to XML-Schema [Bird et al, 2000]. It has also been suggested that the core features of XML-Schema be formalised into a concise and precise grammar notation, such as those commonly found in formal language theory. Work by [Mani et al, 2001] typifies this. Here, the resulting formalism is compared to the ER model and suggests that the Extended ER model, in particular, can be mapped onto the formalised grammar notation.

In general, the research community recognises the usefulness of the markup language approach in semi-structured environments, but at the same time recognises the need for reconciling and integrating this with traditional modelling techniques. This notion, along with the fact that proposals to accommodate schema evolution in XML-Schema are forthcoming, collectively culminates towards an approach that can contribute towards building evolvability into structural specifications. We now consider this in further detail.

As discussed above, XML documents in isolation have proved successful for modelling evolving application domains, but require the inclusion of XML-Schema in order to provide a more complete foundation that is also capable of modelling and specifying the semantics and constraints of the underlying domain. Costello and Schneider [Costello and Schneider, 2000] address the issue that XML schemas must be designed to be evolvable as '*any (internet-related) system that fails to recognise and accommodate both chaos and order is less likely to succeed*'. They list the following factors as characterising the requirements of an evolvable XML schema:

- Addition of new elements/attributes to meet a new requirement and ways to mitigate the impact of such changes. (Unfortunately, dropped elements/attributes and restructuring changes are indicated as impacting systems using the schema and will have to be dealt with outside of the evolvability mechanisms of the schema itself).

- No "lock-step" upgrade of applications: Mechanisms must exist by which an application can obtain a *view* of an instance document that corresponds to the version of the schema it was designed for. This essentially corresponds to the notion of an external view in the traditional ANSI/SPARC database architecture.
- Managing new requirements by using an *open control model.* In particular, this implies that an XML schema declared to have an open content model, allows an XML instance document to have any well-formed XML intermingled with the elements already defined in the schema. Systems should therefore be able to respond quickly in a changing environment.
- Schema evolution using refinement can allow for a systematic and engineered approach for managing schema evolution. Here, a new schema can be created by importing and extending the original schema.

[Costello and Schneider, 2000] also document the syntactic means whereby the relevant parts of an XML schema are augmented to indicate that open content is desired. In essence, this involves the incorporation of a "<any>" flag before and after those schema elements where additions are likely.

In summary, it would therefore appear that this, together with research that is able to formally map XML schema structures to traditional formal techniques, collectively holds promise for a framework geared towards the following:

- Provision of an "open" format for exchanging details concerning structural properties of application domains. This also facilitates potential to support "mixed" modelling in the sense that both (traditional) structured data as well as unstructured textual-type data can be accommodated.
- A modelling paradigm that has formal underpinnings (mappings) to more established conceptual data modelling techniques that are able to support reasoning and consistency checking. However, it should be noted that much of the research concerning this is relatively recent. Approaches will in all likelihood need extensions to be able to cope with further semantic modelling requirements. For example, when compared to ORM, Bird et al. [Bird et al, 2000] cited XML-Schema as lacking the ability to cater for multiple inheritance and certain exclusion constraints.

Further research is required to determine the approach whereby the latter may be best facilitated. The requirement for some form of meta-model, which is able to govern the evolving schema, is also important. This issue will be revisited in section 4.3.1 where schema evolution requirements are discussed.

4.2 Design – Requirements and Solutions

Although difficult to draw distinct boundaries between phases of the SDLC, the design phase of an information system is characterised as providing intermediate models and frameworks to assist developers in progressing from semantic entities and relationships, as well as functional descriptions of desired behaviour, to implementations on some sort of execution platform.

In section 2.1.3 we considered the notion of software architectures as a means for guiding the "programming-in-the-large" perspective. Here, developers view the structure of the system in terms of components and their interconnections. The abstraction level of the components varies over the design phase and may deal with conceptually-oriented views in early stages, to more module-based and implementation-dependent artefacts in later, lower-level stages.

Most literature tends to focus on the early design phase as a stage where evolution, in an architectural sense, should begin to be dealt with. The following work supports this:

- Riebisch and Philippow [Riebisch and Philippow, 2001] note that the accommodation of new requirements into an existing product line typically degenerates the original software architecture that served to specify the system design. Practice has also shown that non-technical organisational factors contribute to this. These include support for human abilities, e.g. understanding solutions, mastering complexity, thinking at higher levels of abstraction, and detecting deficiencies. Better levels of understandability to developers are therefore needed along with improved tool support that can help reduce mistakes made during adaptation of a software architecture.
- Van Gurp and Bosch [Van Gurp and Bosch, 2001] have suggested that current design notations lack expressiveness and that many concepts used during the

design phase are represented implicitly, leading to maintenance difficulties. In particular, object-oriented design is criticised for "hard-wiring" interactions between objects and is not amenable to accommodating changing business rules. A more conceptual separation of concerns is sought in terms of larger architectural components, as opposed to just isolating smaller pieces of code.

We therefore begin by considering a conceptual approach for supporting evolution in terms of the interactions between architectural components, and motivate why this is a key area for addressing evolution.

4.2.1 Accommodating evolution in design architectures

Andrade and Fiadeiro's [Andrade and Fiadeiro, 2001] research has recognised that organisations require business and technology architecture whose components can be added, modified, replaced and reconfigured. Component-based development has often been proclaimed to deliver an approach that can indeed deal with the volatility in business and technological environments. Hopkins [Hopkins 2000], for instance, claims that '*software developers have long held the belief that complex systems can be built from smaller components, bound together by software that creates the unique behaviour and forms the system. Ideally, a new system can be built using mostly predefined parts, with only a small number of new components required ... In a well designed system, the changes will be localised, and the changes can be made to the system with little or no effect on the remaining components*'. Andrade and Fiadeiro [Andrade and Fiadeiro, 2001] however recognise, through development experience in banking domains, that *interactions* and *architectures* in particular, are at the core of the problems that need to be addressed before component-based technology can sufficiently accommodate evolution. The major issue here concerns the fact that it is not changes to the computations performed by the components that are required, but changes to the way in which they interact. Furthermore, the global behaviour of a system is a product of local behaviour of components and the ways in which the components are interconnected.

The solution that Andrade and Fiadeiro [Andrade and Fiadeiro, 2001] propose bears similarities to Saake et al's [Saake et al, 2000] work on modelling adaptive

information systems. This was discussed in section 4.1.2.1 which considered that evolving behaviour could be formally specified through a set of axioms, the contents of which are controlled through *mutators*. Here, we are concerned with a more global level of interaction between system entities and need to consider issues that are more relevant to design. Andrade and Fiadeiro [Andrade and Fiadeiro, 2001] recognise the problem of coding interactions into system components as a key contributor to evolution difficulties. Object-oriented methods are also criticised in that interactions are usually coded in the way messages are passed, features are called, and objects are composed. The end result is often an intricate mix of spaghetti-like structures where interactions are not explicitly revealed. The solution therefore lies in externalising component interactions by making them "first-class" entities. Systems can therefore exhibit their configuration structure explicitly and thereby provide a handle on the architecture for dealing with change. An overview of Andrade and Fiadeiro's [Andrade and Fiadeiro, 2001] *Coordination Contracts* as a new modelling primitive for managing information system evolution follows.

4.2.1.1 Coordination Contracts – Enhancing Design Architectures

Change is more easily perceived at the application domain level, suggesting an *abstract* component model. A mechanism for enabling evolution over such a compositional structure is therefore considered, with the following enhancement to component-based development in particular:

- Provision for explicit representation of coordination mechanisms that regulate the way components behave and interact.
- Enabling of the systems to support evolution through the reconfiguration of the coordination mechanisms in place.
- This reconfiguration should not interfere with the way computations performed by the individual components are programmed.

In essence, the proposed mechanism, or *coordination contract,* is defined in the sense of a UML association class. Figure 4.5 below indicates how such a contract might be defined.

```
contract <name>
        partners <list-of-partners>
        invariants <the relation between the partners>
        constants
        attributes
        operations
        coordination <behaviour superposed by the contract>
        behaviour <local behaviour of the contract>
end contract
```

Figure 4.5 – Definition of a coordination contract

The important aspect concerns the *partners*, *invariants*, and *coordination* definitions. *Partners* specifies a collection of classes that may play a role in the contract. The actual instances of the partners, which may ultimately become coordinated by instances of the contract, are determined through a set of conditions specified as *invariants*. The behaviour that is required to be superposed over that of the partners is identified under the *coordination* definition in terms of trigger/reaction clauses of the form:

```
<name>: when <condition>
        do <set of actions>
        with <condition>
```

The intuitive semantics of this coordination is as follows:

- Conditions under *when* establish the trigger of the clause and may take the form of actions or state changes in the partners.
- The actions under *do* identify the reactions to be performed, and typically takes the form of actions of the partners or actions local to the contract itself.
- Under *with*, further constraints on the actions stipulated under *do* are specified, i.e. preconditions.

For example, in a banking scenario we might envisage the following:

```
contract Traditional Account
        partners x: Account; y: Customer;
        invariants ?owns(x,y)=TRUE;
        coordination
                tp:     when y.calls(x.Withdrawal(z))
                        do x.Withdrawal(z)
                        with x.Balance() >= z;
end contract
```

Figure 4.6 – A coordination contract for a conventional bank account

In the above example, contracts regulate only a specific class of interactions between customers and accounts: those that have subscribed to the contract *Traditional Account.*

Andrade and Fiadeiro [Andrade and Fiadeiro, 2001] indicate that the approach is backed by a formal mathematical basis for coordination, stemming from work relating

to the categorical semantics of architectural and coordination principles. This also underpins their current development of architectural configuration languages that are aimed at assisting in the process of controlling or programming the evolution of systems. This will also include logical mechanisms for reasoning about possible interactions of components. Further research is also directed towards suitable implementation mechanisms for the approach. Exploitation of polymorphism and subtyping in object-oriented programming languages are suggested.
Andrade and Fiadeiro's [Andrade and Fiadeiro, 2001] work might also be considered as a particular instance of a more general solution towards dealing with evolution in software architectures.

We now consider that the structure of a system, in terms of its constituent components and their interconnections, may itself need to evolve. This would typically be the case where requirements and concerns have changed to an extent that has exceeded the scope of the original design's ability to accommodate change and the designer's original anticipated areas of evolution.

4.2.2 Evolving a Software Architecture

Architects may try to anticipate the types of future modifications to an architecture and design it accordingly. Unfortunately, unanticipated changes are still likely. A need arises for supporting architectural evolution such that detection of incompatibilities, inconsistencies, and conflicts during unanticipated evolution is catered for. Failure to do so results in the problems of design erosion – this was discussed in section 3.1.3.

Mens et al. [Mens et al, 1999] propose so-called *reuse contracts* towards a framework that is able to deal with design- as well as run-time or dynamic evolution. In the latter case, changes may either be triggered by the current state or topology of the system, or given by the reuser on a more ad-hoc basis.

A reuse contract essentially consists of a *provider clause* and a *reuser clause* that are related by means of a *contract type*. The provider aspect specifies the properties of an

evolvable software artefact that can be relied upon by other dependent artefacts. The evolver part specifies the modifications that are made to these properties. The *contract type* indicates the exact kind of modification that takes place. Basic contract types include *extension*, and *cancellation* of elements, while relationships between elements can be added or removed through *refinement* and *coarsening* contract types. Contract types may also consist of compositions of these basic types.

Instead of distinguishing between the conventional notions of architectural *components* and *connectors*, Mens et al. [Mens et al, 1999] generalise everything as an architectural element where:

- Elements have external *gates* allowing them to be linked to other elements
- Elements may be primitive or composite. Composite elements may themselves constitute an entire architecture.

These elements are then subject to the contract types *extension*, *cancellation*, *refinement* and *coarsening* – establishing a formalism to reason about the evolution of architectures.

In particular, a basis is established for detecting architectural conflicts. Mens et al. [Mens et al, 1999] state that '*these conflicts will not only occur when the same architectural part is modified in different ways by different evolvers, but can also be used to check compliance between an architecture and its underlying implementation. If the implementation evolves in ways not supported by the architecture, a conflict will be detected. In this way, the problem of architectural drift can be tackled*'.

The important aspect to note about this work is that it aims to contribute to the larger research effort of providing *integrated* support for unanticipated evolution during the entire SDLC, ranging from requirements to implementation and maintenance stages.

To complete the discussion of contributions to different phases of the SDLC as regards evolution, we next consider solutions concerned with implementation and maintenance level stages.

4.3 Implementation and Operation

Designing systems, in such a way that they are able to absorb and dynamically accommodate changing requirements, can help establish implementations that are not overly compromised or disrupted, as can be the case when re-engineering is required in order to realise the required changes.

However, as was introduced in section 3.1.4, once a development has reached its implementation and operational phase, additional factors become relevant and typically include:

- Disparate technologies being called upon to realise a working system. These include database systems, operating systems, communication and network systems, as well as user-interface management systems (UIMS) and the application program itself. Interdependencies between these can be complex, and detract from the conceptual and design-level notions that allowed developers to focus more on the application itself and not the complexities imposed by the implementation technology.
- A potentially large store of programs and data that are long-lived, and often concurrently accessed, now requires management and careful consideration, especially when facing any form of requirements evolution. Consequences of change must now be properly propagated, in addition to ensuring semantic integrity of the system relative to a higher-level conceptual specification.
- Pressures for the system to be able to be integrated with other application systems, such as in the multidatabase sense, or to be able to deliver services to various client types. A typical example of the latter includes Web-related initiatives where information can be easily acquired and analysed as well as used to drive change. This is the case in e-commerce type applications where applications can be expected to dynamically evolve to suit customer requirements.

Research has identified the following areas as contributors to solving evolution problems at the implementation and operation phase:

- Progress in techniques to deal with the schema evolution problem. (Section 4.3.1)
- Meta-programming, and in particular linguistic reflection, as a means for implementing generic specifications and providing an ability to accommodate

change without resorting to highly interpretive approaches or ad-hoc restructuring methods. (Section 4.3.2)

- Persistent Application Systems (PASs) research, with emphasis on orthogonal persistence as a means for overcoming the obstacles imposed by using "disjoint" technologies for realising an information system. (Section 4.3.3)
- Meta-modelling architectures, and in particular the Object Management Group's (OMG) four-level meta-modelling architecture. (Section 4.3.4)

Each of these areas are considered in turn, followed by a discussion on the PJama project (Section 4.3.3.2) and the OMG's Model-Driven Architecture (MDA) (Section 4.3.4.1). The former is a research project concerned with creating an orthogonally persistent Java while still addressing issues posed by evolution. The latter project claims to provide the foundations for building a variety of automatic and semi-automatic software maintenance tools.

4.3.1 Schema Evolution: Current Research and Related Work

Schema evolution was introduced in section 3.1.4.1 and is primarily concerned with the ability of a database system to accommodate modifications of the database schema without loss of existing data. The major research directions are directed towards object-oriented database systems and focus on the following:

Preservation of Semantic Integrity

This concerns the maintenance of the integrity of a schema in terms of the object model, but can also be extended to include support that assists maintainers in ensuring that any change is indeed consistency preserving and correct with respect to the underlying domain. The first requirement has been well studied in work relating to the O_2 database project described in [Zaniolo et al, 1997]. Schema changes are effected in O_2 by modification primitives which include facilities for:

- Modifications to class attributes (creation, deletion, renaming, and modification of the attribute domain)

- Modifications to class methods (creation, deletion, renaming, and signature modification).
- Modification to the class inheritance graph (creation and deletion of superclass/subclass relationships).
- Modifications to classes (creation, deletion and renaming).

A set of invariants and rules for maintaining these invariants over schema changes are also defined. For example, the *class hierarchy invariant* states that the object class hierarchy must have one root and must be a connected directed acyclic graph (DAG) with distinct class names. The set of accompanying rules concerned with *class hierarchy manipulation,* address the aggregation and deletion of inheritance relationships between classes as well as the creation and removal of classes. These also call on a set of *multiple inheritance rules* to resolve any conflicts relating to definitions in subclasses.

Requirements for semantically richer techniques in order to maintain consistency have also been identified. These deal with issues beyond structural consistency. Approaches to this generally involve incorporation of domain-specific knowledge into the schema in a formalised manner. This promotes the application of sound reasoning techniques to help guide and constrain the application of schema evolution operators. Franconi et al. [Franconi et al, 2000], for instance, advocate an approach that extends the object-oriented model in terms of an encoding that promotes the reduction of reasoning problems to corresponding description logics reasoning problems. Chen et al. [Chen et al, 1995] suggest the use of a domain meta-model as a framework for guiding evolution in accounting database systems in particular. By using a domain model, potential target schemas can be suggested. Domain specific heuristics are also included to guide the choice of a sequence of operators to evolve the current schema to the potential target schema.

Compound schema changes

Current systems are generally limited in that they only support changes local to individual types within a schema and thereby limit the richness of changes the

database administrator can perform. The need for tool support is especially relevant here, in order to achieve a more automated evolution of what would otherwise become a tedious and error-prone task.

Pons [Pons and Keller, 1997] and Lerner [Lerner 2000] have proposed solutions. The former suggests an approach where a compound type change is decomposed into simpler well-defined primitives. However, identification of a correct sequence of primitives to collectively realise a compound change has proved difficult, leading to necessary human intervention. Lerner suggests a more algorithmic and semantically-based approach by developing transformers that are able to infer how types have changed (given both old and new definitions of the type). A promising tool (TESS – Type Evolution Software System) has been developed, but is limited to dealing with a type model that does not have inheritance.

Change propagation to database instances

Traditional approaches have involved changes made by the database administrator being immediately propagated to the data. Although ensuring that the entire database is in state consistent with the new schema, the technique generally results in the database being unavailable and encourages a centralised schema change operation – an unattractive solution for systems demanding high availability. Subsequent approaches have suggested a lazy or deferred mechanism for converting the data only when required. Roddick [Roddick 1995] lists the following advantages:

- Changes to the schema can be made more rapidly – improving availability.
- Data are changed only when required, and thus the identification of obsolete data is not required on instantiation of the changed schema.
- The immediate withdrawal of a schema change operation is possible without effect. Furthermore, compensating schema changes may result in no physical data changes at all.

Both immediate and deferred conversions are supported by the O_2 database system. Unfortunately, there is now a data access overhead imposed on the system. A study in [Zdonik 1997] indicates that while the approach is feasible on small databases or on systems where availability is important, real-time applications requiring predictable

response times, as well as large databases faced by a limited number of schema modifications, appeared to be better served by an immediate transformation method.

A third approach, namely that of versioning by view-methods, has received much interest in recent research. Objects are not physically transformed and are instead presented via some emulation mechanism in order to make them appear as adhering to a new schema. The technique is especially attractive for application compatibility and is discussed below.

Application Compatibility

The need to support legacy applications, "as if nothing had happened", is a problem often faced in industry. *Schema versioning* extends the schema evolution problem by requiring that the system provide access to all data, both retrospectively and prospectively, through user definable version interfaces. Recent solutions vary and include the following:

- *View-based support*: Work by Ra and Rudensteiner [Ra and Rudensteiner, 1997] in particular, represents the extent of progress as far as integrating schema evolution with view facilities is concerned. Unlike earlier versioning approaches, potential now exists to overcome problems related to storage overhead for redundant objects. Their Transparent Schema Evolution (TSE) system provides a means for users to specify schema changes to their personal (external) views, rather than directly to the shared base schema. The evolution neither affects other views nor existing application programs. Furthermore, the new views may be capacity-augmenting, which in turn requires augmentation of the global base schema and some database reorganisation at the instance level. Future work is directed towards support for more complex schema evolution operations (e.g. partitioning and coalescing of classes), as well as towards addressing performance issues surrounding the propagation of updates through chains of dependent classes. Ra and Rudensteiner [Ra and Rudensteiner, 1997], however, state that the complexity and overhead of maintaining many separate views can become excessive, necessitating that the conversion of legacy applications be reconsidered.

- *Temporal Database Approaches*: Roddick [Roddick 1995] defines temporal data models as being concerned with the accommodation of the inherent temporal nature of the object world, in addition to the time-dependent recording of facts relating to this object world in a database system. They lend themselves to supporting historical queries by allowing the interrogation of old schemas, as well as the interrogation of old data should temporal support be extended to the underlying database objects as well. Extending temporal support to the management of schema objects (meta-data) has also proved to lend itself to environments such as CAD and software design where design histories are now traceable.

  In general, the resulting complexities, performance, and storage overheads require careful consideration before committing to an implementation of this sort.

Non-versioning approaches are also prevalent in achieving application compatibility. Instead of attempting to avoid change, they either guide the developer through the program parts requiring change by means of compiler warnings of some sort, or are more advanced in terms of supporting some form of program restructuring. Reflection-oriented techniques are useful as they allow their own program structures to be altered from within. These are now considered in further detail.

4.3.2 Reflection

Meta-programming encompasses concepts such as *reflection*, *introspection*, *intercession* and *reification*. Gabriel et al.'s [Gabriel et al, 1993] definition of these terms reads as follows:

'*Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification.*'

It therefore becomes evident that any attempt at delivering a programming environment supporting reflection will need to consider the effect on compilation and loading mechanisms. Type checking, and the provision of a meta-level mechanism providing description of types that support the reification requirement, are also needed.

Linguistic reflection, in particular, has received much interest. Stemple et al. [Stemple et al, 1992] define linguistic reflection as being the (introspective) ability of a running program to generate new program fragments, and to then integrate these into its own execution (thereby achieving intercession). Program behaviour is therefore dynamically modified.

According to [Atkinson and Morrison, 1995] two mechanisms for realising linguistic reflection have evolved:

- *Compile-time linguistic reflection*: This allows the user to define generators which produce representations of program fragments. These generators are executed at compile time. Their results are then type checked and made part of the program being compiled.
- *Run-time linguistic reflection*: This is more concerned with the construction and binding of new components with existing components in an environment. Here, a compiler that can be invoked dynamically is required, in addition to a dynamic incremental loader. Type checking occurs in both compilation and binding phases.

The technique has proved to be an effective component in systems addressing evolution. Atkinson and Morrison [Atkinson and Morrison, 1995] consider it an important contributor in persistent programming environments. In particular, they consider *type-safe linguistic reflection* (where type-checking constrains the allowed output of reflection) as extending the data modelling capability of the type system. This leads to an ability to implement highly abstract specifications, such as those used in query languages and data models, within a strongly typed programming language. Furthermore, a means of dealing with continual changes in data-intensive applications is provided. As an example, Kirby et al. [Kirby et al, 1997] have realised benefits for a weather monitoring application. Here, the schema describing the incoming data needs to dynamically change to suit the needs of the application, and by using

linguistic reflection, they are able to alter the schema in order to record any new structure discovered by applications in the data, programs or meta-data.
We discuss persistent programming environments in further detail in section 4.3.3, and the PJama project, in particular, as an implementation level approach that can cope with evolution.

Lastly, recent proposals to deal with software maintenance and evolution, such as [Brooks et al, 2001] and [Bezivin and Ploquin, 2001], recognise reflection as part of as part of a meta-modelling based approach where aspects implicit in system code are reified to become first-class meta-data objects. We explore this work further in section 4.3.4.1.

Having considered database schema and application evolution concerns separately, the next sections now address the evolution problem on a broader scale, i.e. in the sense of aiming for a complete and coherent solution for the implementation phase.

4.3.3 Persistent Application Systems and Orthogonal Persistence

As was introduced in section 3.1.4.3, the development of any system that is characterised by long-lived, concurrently accessed, and potentially large bodies of data and programs, involves the employment of services from operating systems, database systems, user-interface management systems (UIMS), communication systems, compilers, etc. Unfortunately, variations in naming, type and binding schemes prevail, as well as differences in recovery, concurrency and transactional behaviour. This creates significant complications for the programmer, in addition to the need to maintain translations and mappings between the different components.
The fact that database and programming language communities have followed different development philosophies, in spite of having to provide many similar services, is often cited as one of the key problems in attempting to improve the current state of affairs.
Persistent Application Systems (PASs) research is aimed at addressing the fact that the *application,* as such, typically outlives its individual components and

implementation technology. Healthcare systems, CAD/CAM systems, scientific databases and environmental modelling systems, are all common examples of PASs. Different implementation architectures have been proposed for persistent applications, including the combination of data models and programming languages, as well as the necessary extensions to either databases or programming languages : more complete type and computational facilities for the former, and persistence capabilities for the latter.

In particular, Atkinson and Morrison [Atkinson and Morrison, 1995] clearly motivate that *orthogonally persistent object systems* represent the most likely approach towards realising an environment where the total composition of services (ranging from data definition and operations, to integrity, concurrency and distribution) is supported through one coherent design. They also motivate that '*the construction of persistent systems is made considerably easier when the whole computational environment is persistent. In such an environment, programs and processes may be regarded as data and manipulated in the same manner, allowing transformations traditionally regarded as being performed by a separate mechanism to be executed within the persistent environment.*' In doing so, a basis also exists to simplify system maintenance and evolution.

We begin by describing the notion of orthogonal persistence in further detail and explain why it serves as a foundation for managing evolution. This is followed by a brief overview of the PJama project : a promising realisation of an orthogonally persistent object system based on the Java programming language.

4.3.3.1 Orthogonal Persistence

The term *persistence* specifically concerns the support of data values for their full life time however brief or long these may be. This can range from transient results in expression evaluation and local variables, through to data that outlives versions of a persistent support system. There is a division in this range, where the former part is typically serviced by programming languages, and the latter by databases and file stores. *Orthogonally Persistent Systems* aim to treat data values independently of their longevity, size or type.

Realising this has involved consideration of how programming languages must evolve to cater for persistence, as well as how existing persistent systems (such as object-oriented database systems) might be improved, beginning with the elimination of the impedance mismatch problem.

An environment supporting orthogonal persistence would need to adhere to the following principles [Atkinson and Morrison, 1995]:

- *Principle of persistence independence*: The semantics and form of a program is not changed by the longevity of the objects to which it is applied. Developers are therefore freed from the programming overhead of moving data between long-term and short-term stores.
- *Principle of data type orthogonality*: Any data object of any type can be made persistent. Data modelling is simplified as long-term forms of bulk data types need not be separated from short-term forms.
- *Principle of persistence identification*: The means used to identify and provide persistent objects is independent of the Universe of Discourse of the system. In particular, the strategy of *persistence by reachability*, where an object is made persistent when it is reachable from another persistent object, satisfies this principle.

As indicated in [Connor et al, 1994], orthogonally persistent programming systems allow all data (short and long-term forms) to remain under the control of a single persistent programming system for their entire lifetime. More specifically, the following two factors help establish the basis for an environment conducive to supporting software maintenance and evolution:

- Protection mechanisms are provided over the whole environment through the use of a single enforceable programming model.
- Referential integrity is preserved over the entire computational environment for the lifetime of the PAS. "Secure links" are maintained among the data, meta-data (schema), and program entities. In particular, Atkinson and Morrison [Atkinson and Morrison, 1995] indicate that '*the referential integrity of an object means that, once a reference to an object in the persistent environment has been established, the object will remain accessible via that reference for as long the reference exists*'.

This support clearly assists in change management, where all consequences of a change must be properly propagated (to both data instances and programs). Furthermore, the consequences of change can be better understood in order to avoid unnecessary changes being made and provide better support to automated systems attempting to implement change.

The following maintenance and evolution mechanisms are now better facilitated:

- *Incremental evolution*: Early database methodologies generally advocated that schemas to "represent the enterprise" be designed first, followed by the addition of data and programs. This resulted in schemas being considered as relatively static components of a system. However, incremental development is more feasible where portions of an enterprise are understood, a corresponding design developed, and construction initiated. Orthogonally persistent systems aim to support evolution of program, data and meta-data (schema) by the same mechanisms. Here, all are considered equally and more flexibly and may be static or dynamic, or large or small, depending on the PAS under construction.

  Simply put, programs, data and meta-data evolve in tandem and in increments corresponding to the progressive understanding of the enterprise. There is no bias towards program being more or less incremental than types.
- *Hyper-programming*: In an integrated persistent environment where program, data and meta-data are manipulated by the same mechanisms, programs can be constructed in such a way that objects accessed by the program may already be available at the time the program is composed. Bindings (links) to these objects can now be included as opposed to traditional textual descriptions of where to find persistent values. Such a program is called a *hyper-program*. As discussed in [Connor et al ,1994], the hyper-programming concept provides a technique for representing all executable programs, effectively establishing links between executables and their corresponding hyper-program source representations. This allows a compiler, for instance, to record which programs use which parts of a schema.
- *Linguistic Reflection*: As demonstrated in a dynamic weather monitoring application proposed by Kirby et al. [Kirby et al, 1997], orthogonally persistent

systems overcome the ordering difficulty imposed by traditional database systems. As regards the latter, the database is constructed by first defining the meta-data (schema) and then initialising data in accordance with the meta-data description. Where data with a new structure is required, the process is repeated in the same order. Thus, programs discovering new structure about existing data have an ordering difficulty: the new structure must exist before the program runs but this is only discovered during execution. In an orthogonally persistent environment, computations over the meta-data, data and programs are possible. The basis is established for new meta-data, programs and data to be bound into the executing system, thereby facilitating linguistic reflection.

To conclude this section we briefly discuss the PJama project – a research system geared at demonstrating the claim that orthogonal persistence is indeed a better application programming technology.

4.3.3.2 The PJama Project

The PJama project developed at the University of Glasgow, Scotland, in conjunction with Sun Microsystems, is focused on developing an orthogonally persistent version of the Java programming language. In doing so, a vehicle is established to test the claims that orthogonal persistence does indeed yield an improvement in application programming technology.

Aside from its popularity as a language for enterprise application implementation, the following characteristics motivate Java as the language choice for the PJama project [Atkinson et al, 1996].

- Strong typing
- Single inheritance
- Object-oriented model
- Automatic space management
- No explicit manipulation of pointers
- Validations to improve security, precision and productivity.

A detailed account of the recent progress in the project can be found in [Atkinson and Jordan, 2000]. We briefly survey the state of affairs, with respect to the requirements listed in [Atkinson and Jordan, 2000], in order to give the reader an appreciation of some of the challenges involved.

- *Orthogonality*: In establishing orthogonality, the current support surrounding the use of *java.jdbc* and *org.omg.corba* class libraries (providing application programming interfaces to external databases) is hampered by the fact that they are inherently transient from the persistent program point of view. To overcome this, effort has been spent on investigating how the state of these "external computations" can be captured and how correct resumption of program threads (that persist through their reachability) can be supported should execution be interrupted. In general, resumption of such program threads is noted as a fundamental difficulty due to the intertwining of the *java.lang.Thread* class with the underlying Java Virtual Machine (JVM).
- *Persistence Independence*: Here the Java language must remain completely unchanged (syntax, semantics and core classes) so that imported programs and libraries of classes work correctly. This has significant benefits for code reuse, allowing programmers to move freely between persistent and standard versions of the Java platform. Aside from requiring clarification on the semantics of code resumption on the Java platform, persistence independence is noted as essentially being achieved.
- *Durability*: Durability, where loss of data is avoided due to software, platform or hardware failures, is presently limited to small development environments where "disruptive" off-line archives can be taken. This is clearly a problem in large scale enterprise deployments where continuous operation is mostly required. Support for evolution and migration technologies is also regarded as a pre-requisite – the need to discard stores so that an application can change to meet new requirements will result in a failure of durability.
- *Scalability*: Developers must be protected from the effects of scale, whether the implementation is intended for hand-held devices or terabyte-size enterprise stores. The review in [Atkinson and Jordan, 2000] indicates experiments on 10

gigabyte stores, but with future PJama projects requiring between 2 and 5 terabyte stores.

- *Evolution*: Like any persistent store, mechanisms are required to manage change to classes and instances. At present, the PJama evolution technology is focused on development-time evolution where developers are constantly making small changes to experimental or prototype persistent stores. The following characterise the nature of this support:
  - *Specification of class changes and instance transformations*: The developer can specify changes to class hierarchies as well as renaming, deletion and insertion of classes. Default transformation code exists to transform old instances of a class to new instances of that class or some related class. However, the developer may also use the Java programming language to specify arbitrary computations to obtain new values for instances where classes may have changed format.
  - *Mutual consistency*: Mutual consistency across classes is now achievable. The promotion of classes to first class persistent objects is especially useful for controlling consistency in terms of behaviour. For example, methods deleted from one class, but still called from another, can be detected. The so-called *Persistent Build Technology,* described in [Dmitriev, 2000], combines class evolution and recompilation of class sources, thereby keeping track of changes to application classes, and preventing them from being left incompatible after a change.
  - *Atomic execution of transformations*: Support for performing the transformation atomically is available. However, the current transformations run in an off-line environment (i.e. no on-line or concurrent evolution support at present). Object conversions are also complete, as opposed to lazy conversion methods. The latter is cited as adding much complexity to the evolutions support system.

Further research work is focused on support for deployment-time evolution. This is complex, requiring that changes be installed on a customer's implementation without losing any investment in data and programs. Furthermore, interruptions to application availability are mostly unacceptable. Dmitriev [Dmitriev 2000]

indicates that versioning techniques, specifically those concerning the long-term co-existence of multiple versions of individual classes or collections of classes (schemas), are complex in the PJama environment and any attempt would require considerable effort. This is associated with the present restriction that PJama does not support multiple applications running concurrently over the same store.

Lastly, developers would also require the integration of evolution management into development tools. This assists in easily managing and changing persistent stores during design.

- *Other challenges*: Migration to other platforms, i.e. advances in the Java platform and use of new persistent store management platforms, requires careful consideration so as to preserve orthogonal persistence. Although current research is centred on off-line migration, the need for incremental evolution for large-scale systems must be accommodated. Related to this is the accessing of "external computations" (e.g. JDBC interface) so that interaction with autonomous components does not compromise the system's integrity. PJama extends the Java Remote Method Invocation (Java RMI) standard to combine persistence with distribution. However, Atkinson and Jordan [Atkinson and Jordan, 2000] indicate that solutions that are scaleable and more supportive of class evolution require further research.

  Further work is also required to allow many applications to run concurrently against the same store. At present, developers would have to manage this themselves, leading to solutions with excessive locking, poor performance and deadlocks.

The next section explores a meta-modelling approach, where the notion of conformance to a meta-model, aims to provide developers with an abstract model-driven implementation framework that is independent of the underlying execution platform.

4.3.4 A Meta-Modelling Approach

As systems become more complex, it has been recognised that methodologies that rely on increasingly more abstract mechanisms are required in order to ensure

manageability, especially in the face of evolution of implementation platform requirements and changing user requirements. Meta-modelling approaches generally employ a framework where a hierarchy of abstractions are used, such that evolution of any one layer is constrained (and guided) by the later above it.

The four-layer meta-modeling architecture employed by the OMG is characterised as follows:

- *Meta-meta model layer*: Provides a language for defining meta-models and can be thought of as a meta-grammar such as EBNF. The OMG's Meta-Object Facility (MOF) exists at this layer as a means for managing meta-models in a standardised manner. It defines the essential elements, syntax, and structure of meta-models that are used to construct object-oriented models of discrete systems.
- *Meta-model layer*: Comprised of the descriptions that define the structure and semantics of meta-data, i.e. descriptions for the model layer. The notions of class, attribute, operation and component are specified here.
- *Model layer*: This defines the language for specifying information domains. Elements like Student, Teacher and Course classes are domain-specific examples of elements belonging to this layer.
- *Instance layer*: Comprises instances of the elements defined in the model layer; for example Student, Teacher and Course objects belonging to the classes defined previously.

According to [OMG-MOF, 2000], this four-layer architecture has a number of advantages:

- Assuming that the meta-meta model is rich enough, it can support most if not all kinds of meta-information imaginable.
- It potentially allows different kinds of meta-data to be related. (This depends on the design of the framework's meta-meta model).
- It potentially allows interchange of both meta-data (models) and meta-meta-data (meta-models). (This presupposes that the parties to the exchange are using the same meta-meta-model).

For an enterprise's information systems in particular, the need for a global integration framework arises since database, workflow, software process and component management meta-models are often independently defined and independently evolved. Furthermore, by establishing a framework where meta-data objects are reified to become first-class objects, a basis is provided that can support reflection, and hence dynamic configuration and reconfiguration of programs and data. This approach is widely supported in the literature but is receiving significant attention in its application to the OMG's Model-Driven Architecture (MDA) [OMG-Soley, 2000]. This has been primarily proposed as a means for being able to derive code from a stable model, and hence achieve greater independence from underlying implementation platforms. There are also benefits in managing software maintenance and evolution. We briefly consider the MDA vision below.

4.3.4.1 Model Driven Architecture (MDA)

The MDA extends from system specification, dealing with the modelling of business functionality and behaviour, through to implementation where subsequent interoperability issues across different middleware platforms become relevant. The latter is conventionally dealt with through using standard component interfaces across heterogeneous software systems.

The MDA concept adopts a different approach by using formal system models to facilitate interoperability. As indicated in [Poole, 2001], the most significant aspect is the independence of the system specification from the implementation technology or platform. The system definition exists independently of any implementation model and has formal mappings to many possible platform infrastructures such as Java and XML/SOAP.

By using the Unified Modelling Language (UML), Meta-Object Facility (MOF), XML Metadata Interchange (XMI) and Common Warehouse Meta-model (CWM), a basis is established for authoring, publishing and managing models within a model-driven architecture. This includes the following:

- *Platform-Independent Model:* Firstly, in order to realise platform independence, the OMG advocates a Platform-Independent Model (PIM) expressed in UML – a MOF-compliant meta-model. OMG have also created CWM as a standard for representing database schemas. UML serves as the notational basis, but is extended with data warehousing and business analysis domain concepts. This is also used to describe non object-oriented artefacts such as relational, network, hierarchical or XML-based data sources.

  The PIM effectively represents the logical view in which the composition and behaviour of all components are fully specified (without implementation-level details). The intention is that this is then mappable to one or more *Platform-Specific Models (PSMs)* which are again expressed in UML but now contain implementation-specific details.
- *Meta-data exchange*: In order to realise interoperability across different meta-models, a means for exchanging model information is required. The introduction of XMI has provided an interchange format for models (and meta-models) that is based on XML and MOF. XMI effectively defines how XML tags are used to represent serialised MOF-compliant models in XML. A basis is therefore established whereby both metadata (tags) and the instances they describe (element content) can be packaged together, enabling applications to readily "understand" instances via their metadata. This is clearly advantageous in distributed, heterogeneous environments and also enables UML models to serve as the basis for other tools such as code generators.
- *Common services*: The OMG has also recognised that applications rely on a set of essential services, including persistence, transactions and security. When implemented on a particular platform, they also tend to take on the characteristics that restrict them to that platform, or ensure that they work best there. To address this, UML models of these services are presently being constructed for the PIM level. Their functionality and interfaces in multiple middleware targets is also being defined.

Armed with the elements of shared metadata, formal PIM to PSM translations, and a vehicle for exchanging model information, the OMG's MDA vision is set towards facilitating software evolution and maintenance from different perspectives:

- *Portability and interoperability between middleware platforms*: As stated by Soley [OMG-Soley 2000], it is difficult for large enterprises to standardise on a single middleware platform. The most visible environments today include CORBA, EJB, XML/SOAP and .NET. However, it can be assumed that these will evolve and/or be replaced. The platform independence gained by using the MDA approach, helps counter the overhead of expensive and disruptive migrations to newer implementation platforms. Interoperability across platforms is also improved and defined more rigorously.
- *Business models and implementation technologies evolve independently*: The separation between business models and the implementation technology, promote the preservation of the development invested in components when a technology shift occurs. Mappings to different implementation platforms exist in the MDA and can be augmented when new ones are introduced. The OMG will standardise these mappings, while vendors will implement them in their tools, enabling automatic interoperation with or porting to the new platform.
- *Structural (schema) and behavioural evolution is better accommodated*: UML model maintenance, in particular, has conventionally required manual intervention to ensure consistent propagation of change to the associated code. The MDA framework aims to provide a more automated and systematic approach to this via formal metadata definitions that can assist in guiding and constraining evolution. The MDA also aims to incorporate highly generic *core models* of common computing environments, such as *Enterprise Computing* with its component structure and transactional interaction, or *Real-Time Computing* with its resource control requirements. These assist in providing semantic details to evolution mechanisms. Poole [Poole 2001] notes that highly domain-specific metadata that does not fit the generic model, is handled through the use of extension mechanisms that are predefined as part of the generic models (e.g. the use of UML extension mechanisms, such as tagged values, stereotypes, and constraints).

Future MDA visions incorporate the notions of adaptive software and dynamic evolution. In particular, the run-time interpretation of shared metadata is central to the approach – this discipline is termed Adaptive Object Models (AOM). Poole

[Poole 2001] indicates that '*system functionality will gradually become more knowledge-based and capable of automatically discovering common properties of dissimilar domains, making intelligent decisions based on those discoveries, and drawing and storing resulting inferences. In general, "knowledge" is supported by an advanced and highly evolved concept of ubiquitous metadata, in which the ability to act upon, as well as revise, knowledge at run time is provided through Adaptive Object Models (AOMs).*'

This generalised metadata management, authoring and publishing capability holds promise for support of advanced reflection capabilities – both structural and behavioural. The intended result is the production of highly dynamic and self-organising systems. These are then able to act directly on domain knowledge and realise a consistent and complete modification through the system model.

The problem of evolution, in terms of issues and requirements, has been presented. This chapter considered different solutions proposed in the literature, and discussed some of the areas that future research would entail. The next chapter provides a synopsis of the problem, together with the techniques required towards improving the accommodation and facilitation of evolution. A future perspective on supporting evolution coherently and consistently over the SDLC is also considered.

CHAPTER 5 – A SYNOPSIS

This chapter reconsiders the phases of the software development life-cycle in terms of their role in accommodating and facilitating evolution. A brief consideration of how future information system development can deal with the evolution problem concludes the chapter.

5.1 A characterisation of current approaches

In chapters 2 through 4, we discussed the development phases of an information system with particular emphasis on the database (persistent) component of the system. The issue of evolution was essentially considered in terms of two dimensions:

- *Conceptual and Design level accommodation of change*: This concerned how systems can be conceptualised and designed in order to accommodate changing requirements, thereby lessening any later re-coding and re-implementation efforts that may be required to realise the change.
- *Implementation and Operation level accommodation of change*: This deals with the mechanisms and technologies that must be in place to ensure semantic integrity and consistent propagation of changes to all the implementation artefacts.

We now reconsider the software development life cycle (SDLC), the problem of evolution, and the contributions that have been forthcoming to deal with evolution. In particular, their advantages and disadvantages are considered, including their suitability towards the following:

- Facilitating automation of change, i.e. contribution to tool support
- Understandability to developers and other stakeholders (e.g. customers who are involved in the requirements specification phase)
- Implementation feasibility
- Support for ensuring semantic integrity of the system and the ability to ensure consistent propagation of change to all programs and data comprising the system.
- Applicability and scope of the mechanism. For example, does the approach apply only to business systems, or does it hold relevance to scientific database applications as well.

- The nature of the evolution that is supported, i.e. does the approach only deal with a set of anticipated change requirements, or is the approach geared towards delivering a more generic solution where unanticipated change was also catered for.

5.1.1 Requirements Analysis

This phase serves to capture the customer's functional and non-functional requirements for the system. As indicated in section 2.1.1, this included a high-level abstract model of the system in terms of the major relationships and entities, as well as the transformations (actions) that occur in the system. The assumptions on which the system is based, in addition to the anticipated changes in user requirements, are usually documented.

It is the one phase where all stakeholders of the end product have a view on the system that is void of any complex specification and design-level constructs or formalisms. Unfortunately, the following issues are prevalent and influence the evolvability of the system:

- *Lack of foresight*: Future system requirements may, for instance, include the ability of a factory production management system to accommodate new product lines, changes in workflow, etc. Furthermore, any ignorance of the likely need to later integrate the system into a larger federated architecture [1] can weaken and complicate the extent of integration, particularly in reconciling semantic discrepancies. For example, an object-oriented database schema whose classes do not easily generalise into a common organisational class, or set of classes, could prove awkward in terms of reconciling inconsistencies along generalisation/specialisation class hierarchies.
- *Lack of formality*: Natural language is inherently ambiguous and informal, leading to difficulties when reasoning about the system at the specification phase. Improved methods of requirements engineering are sought that can assist analysers in documenting the system within a framework serving both the need for

[1] In the sense of a federated database system where heterogeneous DBMS's are either affiliated via a central global schema or interact loosely via exchange schemas

understandability by non-technical stakeholders, as well as the need to provide a basis for more formal system specification.

Techniques to guide the requirements analysis process can be employed. Firstly, Ghose [Ghose 1999] indicates that *design rationale* (the process of documenting the reasoning process undertaken in designing an artefact) can be useful in requirements engineering, particularly for resolving conflicting viewpoints on the system requirements. This is often the case in industrial software management systems, where organisational management requirements and shop floor supervision requirements tend to conflict.

Although design compromises can sometimes result, the design rationale process can encourage participation towards more complete, more thorough, and more insightful requirements. Evolution-related requirements can also be better revealed and more cautiously considered in context with other requirements.

Secondly, Wu and Han [Wu and Han, 2002] suggest the use of XML-based tools for managing system requirements as well as architectures. XML technology is inherently geared towards exchanging data across organisations and heterogeneous implementation platforms. It would therefore appear that using XML as the format for the information repository for a requirements analysis document is advantageous. In particular, requirements are more traceable – an important consideration should requirements need to be amended or evolved. The framework proposed by Wu and Han [Wu and Han, 2002] is essentially a fixed one, where requirements are captured in terms of stakeholders, goals, assumptions, components, services, quality of services, etc. Although this is feasible towards improving the formality and rigour of the requirements documentation, it also tends to dictate a certain path to establishing the system requirements. This can be awkward and restrictive to non-technical participants who are more at ease with articulating requirements using informal and natural language type approaches.

Hypertext-based documentation management systems have assisted in providing "loose" associations between natural language documentation and structural documentation. References to entities and relationships in the underlying application domain are feasible, although behaviourally-oriented references are more complicated – semantics would need to be carefully considered. Any automated progression, from

a natural language basis to a more structured and formal interpretation, would require Artificial Intelligence (AI) participation. In particular, this would demand production of semantically correct specification-oriented artefacts that define both structural and behavioural concerns. The inherent ambiguity and informality of natural language constrains the feasibility of such approaches.

5.1.2 Specification and Conceptual Modelling

The specification phase is especially important in its role as a transition between more informally-oriented requirements documentation and system design – the latter providing the blueprint for an eventual implementation. Different specification formalisms are used to model the structural, behavioural and control flow aspects of the system. Conceptual models such as the Entity-Relationship approach define structural requirements. Functional specification techniques, ranging from model-driven formalisms such as Z to pseudo-code like descriptions, define behavioural concerns, while data flow diagrams usually represent the transformations and control flow concerns. As most information system applications are centred around the conceptual model, common techniques were described in section 2.2.1, including object-role modelling (ORM) and object-oriented modelling. XML was also discussed as a valuable technique for describing semi-structured data and was also considered further in section 4.1.2.2 as a possible means towards building evolvability into structural specifications.

From an evolvability point of view, the following issues are of particular interest:

Assessing evolvability

As discussed in section 3.1.2.1, different conceptual models can be used to describe the same application domain, but exhibit different evolvability characteristics. *Stability* emerged as a desirable characteristic, and metrics were presented in section 4.1.1 as a means for measuring this quality in conceptual models existing in operational environments. Although useful in assessing the stability of models, the

approach would appear to best serve prototype-style developments, where shortcomings can be addressed and rectified without any serious impact on implemented artefacts and persistent data. It does not explicitly address *how* evolvability requirements are specified.

Modelling evolvable requirements

Behavioural requirements are generally considered to be the most volatile. Business rules in particular are subject to change. In order to accommodate this anticipation of change in the specification, a separation of rigid and evolving parts was suggested and described in section 4.1.2.1. It was also indicated that the approach can be supported by an underlying temporally-based logic, but required the designer to determine the evolving parts from the rigid parts in advance. Constraint is also required, as resorting to a behavioural specification comprised only of evolving axioms creates the problem that everything is possible.

Although generally less volatile than behavioural requirements, the notion of modelling evolving structural requirements was also considered. XML was discussed in section 4.1.2.2 due to its application toward modelling domains with irregular structure where predefined schema structures prove to be awkward. However, the need for structural specifications that are able to support reasoning and consistency checking, in order to maintain the semantic integrity of the system during evolution, are still required. The introduction of XML-schema, described in section 2.2.2, aimed to achieve this for XML-based systems in particular. There would therefore appear to be a tension between having a sound "semantically intact" model, while still being afforded the means to specify content that does not easily conform to traditional rigid schema structures. A likely solution would be the distinction of stable parts from evolvable parts, as was suggested for behavioural specifications. The stable parts could be specified using sound semantic modelling techniques, such as ORM, while "open content" could be specified in terms of XML constructs *appended* to the stable model. Such a solution may be useful in inventory management systems dealing with a large variety of product types. The price, quantity on hand, manufacturer, etc., are qualities applicable to any item. However, it may also be necessary to record information peculiar to certain products. For instance, the features of a digital camera

are entirely different from those of fresh produce! In general, the reasoning and consistency checking in such a scenario would mostly be limited to the stable parts. Any evolvable extensions must merely conform to XML's syntactic requirements.

Uniform specification techniques

It is desirable, especially from a transformational development point of view, to have a specification technique whereby structural, behavioural and control flow concerns are uniformly modelled. As indicated in section 2.1.2, no one method can adequately meet all modelling requirements. The object-oriented Unified Modelling Language (UML) appears to be the most popular approach to capturing both structural and behavioural concerns, with data flow being modelled more implicitly in terms of specifications describing system behaviour. The implementation bias towards object-orientation in particular, is one of the criticisms levelled against UML as a specification phase technique. Potter et al. [Potter et al, 1996] indicate that an early deconstruction of the problem domain into objects, causes a hindrance to the process of considering and capturing system-wide invariants – an issue when reasoning about the system in a formal specification sense.

However, the central role of UML in the OMG's Model Driven Architecture, described in section 4.3.4, tends to indicate that any disadvantages can be outweighed by significant advantages in terms of automated code generation. This approach is discussed again in section 5.2 as a valuable component of future-oriented solutions for coping with the evolution problem.

5.1.3 Design

The progression from abstract and mathematically-oriented specifications to design artefacts is difficult and usually informal. Designer creativity, in deciding on the system decomposition, is usually required.

Software architectures were introduced in section 2.1.3 as a means for assisting designers of large enterprise systems. In particular, frameworks are usually provided,

indicating the decomposition of the system into components and how functionality is assigned to those components.

The problem of *design erosion* was described in section 3.1.3.2. This leads to difficulties for designers in terms of the traceability, understandability, and inter-relationships of design decisions. Furthermore, the fact that traditional design methods encourage the practice of creating a design in advance, causes conflict with the iterative nature of enhancing and augmenting a system design. Van Gurp and Bosch [Van Gurp and Bosch, 2002] indicated that successive iterations tend to erode the design in the sense that the original architectural framework is violated. This has far reaching consequences on the resultant code. Any introduction of external dependencies, such as the interaction with a global variable, would for instance violate the reusability of a component-based design. Although designs realising practices such as abstraction and modularity prove to be more maintainable and evolvable, the factors discussed above suggest that further issues contribute towards addressing evolvability. These include:

- *Expressiveness of design-level representations*: Designs that are more conceptual in terms of the underlying application domain, promote an improved understandability of the complexity of the system. Once again, the virtues of a formal specification that is transformable to design-level artefacts come to bear. In particular, the inter-relationships of components and consistency of the system as a whole can be reasoned about. However, the overheads of formally specifying all aspects of a large-scale information system are considerable, causing developers to resort to building designs without specification-level foundations.
- *Externalising component interaction*: Although component-based design is generally considered to aid evolution, as was discussed in section 4.2.1, it is specifically the interactions between design components that are most subject to evolution. The hard-wiring of these interactions into code contributes to difficulties when needing to evolve the way in which objects interact. The approach of externalising interactions (see section 4.2.1) effectively recognises the need to provide a design, and hence deployable artefact, of a requirement that is inherently conceptual in nature, i.e. the semantics of coordination between entities in the application domain. The benefit offered over conventional object-oriented

designs is that the evolution process, in terms of object interaction, can now be explicitly controlled. However, the extent of evolvability is still capped by the foresight that existed at the time of analysing the underlying application domain, since it is here that evolvable parts are conceptually separated from stable parts.

- *Reasoning about the evolution of software architectures*: As discussed in section 4.2.2, software architectures must themselves evolve in order to accommodate changing requirements and prevent the problem of design erosion. *Reuse Contracts* were then introduced as a means of formalising the modifications that can occur to an architecture. These establish a means to detect conflicts and preserve consistency with respect to the architecture.

The design phase must be viewed as a bridge between early requirements specification and the detailed design phase where program and code are readily producable for the implementation phase. Few large-scale information system developments employ a transformational development style from detailed specification through to implementation. From an evolution point of view, this tends to place the onus on software architectures and design abstractions. These must ensure that expressibility of conceptual level concerns is possible, constructs exist to separate static from evolvable parts, and that the design is interrogatable in terms of preserving architectural integrity.

5.1.4 Implementation and Operation

At the implementation phase, design level abstractions are realised in terms of program and code destined for a particular execution platform. It is often the case that flaws and omissions in early requirements stages are only now revealed. This leads to developers re-iterating through earlier phases to rectify problems – an expensive process in terms of time and also contributing to the likely erosion of the original design. The range of problems increases once a system becomes fully operational and include corrective, adaptive and perfective maintenance as was discussed in section 2.1.5. Furthermore, the recent surge in the internet and other Web-related initiatives such as e-commerce, also require horizontal integration of systems. Personalisation of software, in particular, is considered as a key requirement for e-commerce related

systems. Moreover, any system evolution to meet these requirements should be as dynamic as possible, necessitating the trend towards systems that are "self-organising", in the sense that they adapt to the context in which they are used.

Evolution on established systems is complicated by the existence of persistent stores of data and program, both accessed concurrently by end-users. This, together with the issues presented in the above paragraph, have resulted in numerous solutions being proposed in the literature. Some of these were discussed in Chapter 4, beginning with solutions to the schema evolution problem introduced in section 3.1.4.1. Approaches generally involved the propagation of changes to database instances and required modification of dependent application code, or the creation of emulation-type mechanisms such as views to provide a "virtual" change. Although these techniques are relatively advanced in terms of addressing schema evolution issues, they are also characterised by the following concerns:

- *Legacy-system oriented*: Schema evolution techniques deal with the problem at the operational stage and, in general, tackle the symptoms of evolution as opposed to the cause (e.g. unstable conceptual design). However, they also take cognisance of the fact that established and operational systems generally have availability and application compatibility requirements. Here, a redesign and subsequent re-implementation of database schemas would result in massive disruption.
- *Technically-oriented, complex and specialised*: Many schema- and database evolution mechanisms tend to provide solutions that may be effective but carry significant overhead. An example is Ra and Rudensteiner's [Ra and Rudensteiner, 1997] approach to schema versioning in terms of extended view support. This was described in section 4.3.1. While attractive in providing a solution to address multi-versioning concerns, it is complex in nature and requires maintenance of the dependencies between successive view implementations.

  Other approaches address the issue of semantic integrity of the evolution process, but tend to be limited to particular application domains. The evolution mechanism for accounting-based systems, described in [Chen et al, 1995] and discussed in section 4.3.1, is an example.
- *Manual intervention required and not conceptually-oriented*: Current evolution systems are limited in their capacity to automatically realise compound-type

changes (e.g. merging of object classes), as well as lacking in their ability to preserve semantic consistency. As a result, many systems require manual intervention, or provide the user with a list of suggested approaches for evolving a particular aspect of a schema. Support for maintaining the consistency of dependent applications has either focused on view or versioning solutions to maintain compatibility, or is reliant on compiler-oriented warnings to guide maintainers to affected program parts.

New implementation-level paradigms, such as orthogonally persistent systems (section 4.3.3) and meta-modelling approaches (section 4.3.4), have provided significant progress towards dealing with evolution at the implementation and operational stages.

Orthogonal persistence was noted as:

- Effectively eradicating the impedance mismatch problem.
- Simplifying programming overhead by allowing developers to focus on the application domain, as opposed to how it may be implemented in terms of integrating disparate sub-systems (e.g. database systems, operating systems, communication systems, etc.).
- Automatically promoting propagation of change through meta-data, data and program structures.

At this time, orthogonally persistent systems are mostly limited to research environments, with further progress required in order to make them commercially viable. In particular, the efficiency and reliability of established DBMSs presents a benchmark for orthogonally persistent contenders. The role of orthogonally persistent systems, in terms of supporting evolution in the broader context of the entire software development life-cycle (SDLC), is considered in section 5.2.

Another recent proposal for system implementation concerned the pervasive use of meta-models, such as the Model-Driven Architecture (MDA) described in section 4.3.4.1, in order to establish a high-level abstraction whereby the following are achieved:

- *Design-centred development and platform independence*: Development is centred on the creation of formal design-level models that are compliant to higher-level

meta-models. Mappings to implementation-level constructs are provided and facilitate automatic code-generation should the target platform change. The original design investment is therefore protected.

- *Horizontal integration*: The approach is also centred on exchange of meta-data in order to promote interoperability across different meta-models, be they meta-models describing database, workflow, or component management concerns. This is generally seen as a progression towards more semantically consistent integration, as opposed to the conventional means of only realising interoperability though standard component interfaces. This is, in essence, similar to the aim of orthogonal persistence, where developers are freed from the concerns of interfacing heterogeneous system components. The integration, in the MDA case however, takes place at a low and detailed design-level abstraction as opposed to the implementation phase.

As noted in section 4.3.4.1, the MDA vision is still incomplete, requiring the development of abstract models providing essential services such as persistence, transactions and security.

Finally, orthogonal persistence and the meta-model vision should be seen as complementary solutions towards improving implementation technology that supports evolution. It should, however, be noted that the former encourages a "clean-slate" approach whereby applications are re-engineered into a orthogonally persistent environment. The MDA vision, in turn, offers promise for legacy systems in terms of offering meta-model descriptions on non object-oriented artefacts, such as relational or network-model data sources. However, this would appear to encourage "wrapper-based" solutions. These can sometimes detract from the original conceptual specification of an application in terms of compromised and more restricted functionality.

The next section explores the notion of the specification serving as the core artefact for system development and evolution.

5.2. A future perspective on supporting evolution over the Software Development Life-Cycle

From section 5.1, it becomes evident that the earlier development stages of requirements analysis and specification are vital. The creation of abstract models to represent the underlying application domain provide a framework that is understood by customers of the system, as well as those responsible for developing and maintaining the resulting operational infrastructure. Evolution-related requirements are best catered for at this level and can suggest designs able to meet stability requirements (in the sense of conceptual stability discussed in sections 3.1.2 and 4.1.1). This tends to suggest that a detailed and comprehensive specification would, through a transformational development style, be ideal as the "entry-point" for the implementation of any changes to the system. A sufficiently formal specification meeting Balzer's [Balzer, 1986] requirements for a "good" specification (section 2.1.2), would also model the inter-relationships and dependencies between system components – certainly promising for ensuring the consistent propagation of change and preservation of semantic integrity following an evolution.

Unfortunately, most commercial system developments face the following factors:

- *Pressure to produce deliverables*: Procurers of a new system generally require the rapid production of deliverables in terms of executable system components. This leads to developers spending less time on specification, and more on producing low-level design artefacts that are readily transformable to implementation constructs for a particular platform. These may be prone to instability at later operational phases. Transformational developments from specification through to implementation are also regarded as complex, particularly the verification of transformation steps – tool support may exist, but often requires much user intervention to ensure that the conceptual constructs in a specification are correctly mapped to the different heterogeneous components that ultimately comprise an implementation.
- *Heterogeneous specification methods*: One specification method can seldom meet all requirements. In particular, Object-Role Modelling (ORM) may be used for specifying entities and their relationships, while the functional specification is based on either a model-driven approach such as Z, or pseudo-code constructs.

These factors are generally typical of the disadvantages of any formal methods development.

However, the emergence of mapping technology between specification formalisms, orthogonal persistence, and the meta-modelling oriented MDA, collectively improve the notion of the specification as the core artefact for system development and evolution.

By being able to convert different specification constructs into one homogeneous specification, a basis can be established for further refinement into implementation-level artefacts. Polack [Polack, 1992], for instance, describes a technique (capable of being automated) for formalising an Entity-Relationship model into a series of Z state schemas. Kim and Carrington [Kim and Carrington, 2000] also describe a formal mapping between Z and UML. The progression from a specification to implementation is significantly simplified in an orthogonally persistent environment. The primary reason is that the late design-level and early implementation-level concerns of integrating diverse system components is all but removed.
Current system developments are, however, very reliant on the provision of a software architecture to guide the developer from specification to implementation. Although the MDA approach is primarily targeted at making development more model-based to counter the proliferation of middleware platform changes, the "side-effect" benefit of generation of code from model structures better supports the implementation of requirements changes. Consistency and propagation of changes through to system components on implementation platforms is now better facilitated. Conceptually-oriented development is also more documentation- and hence maintenance-friendly. This would significantly improve the poor documentation and specification artefacts in industrial and commercial environments where high IT staff turnover cannot allow for the maintenance onus to lie on an individual or group of individuals. In essence, the MDA approach would also seem to address both the adaptive and perfective maintenance categories introduced earlier in section 2.1.5. The approach does however constrain development to conforming to MDA meta-models in order to achieve the benefits of simplified maintenance.

CHAPTER 6 - CONCLUSION

Requirements changes, both in terms of application domain and implementation platform evolution, necessitate the study of the design and development of information systems, in addition to the issues involved at later operational and maintenance phases. In particular, the impact of evolution was considered in terms of the software development life-cycle, ranging from requirements analysis stages through to operational stages.

The earlier user-centred and specification-level artefacts emerged as vital components in improving the current state of affairs. Any significant evolution, or maintenance on operational systems, demands documentation that is consistent with the implementation artefacts.

As discussed in section 5.2, orthogonal persistence and MDA-type approaches hold promise for realising a transformational development style, from system model to implementation platform. Such transformations do, however, require a very detailed specification (effectively bordering on design-level constructs) in order to be realised. Future work regarding orthogonal persistence is largely focused on the PJama project, described in section 4.3.3.2. Benefits to be realised by such implementations also include the exploitation of linguistic reflection as a means to facilitate adaptive behaviour. Similarly, the MDA approach is focused on adaptive objects, i.e. software capable of automatic discovery of properties of its environment and adaptation to that environment. Poole [Poole, 2001] indicates that '*our ability to engineer such systems will come largely as the result of our extensive experiences with the use of meta-models and ontologies in influencing system behaviour and decision making. We will eventually learn how to build systems in which a considerable amount of domain knowledge is pushed up into higher abstraction levels. Systems will understand how to efficiently extract and act on that information.*'

The accommodation of change, in terms of building evolvability into system specifications, was also considered. Dealing with the problem as proactively as possible would reduce the maintenance overhead once the system reaches its operational phase.

Section 4.1 discussed the need to measure the stability of conceptual models as a means for gauging the evolvability characteristics of a particular model, i.e. the ease with which the information system can be adapted to changing functional requirements. The metrics presented are most applicable when researching the conceptual models in their "natural environment", the operational business. Contributions from Wedemeijer [Wedemeijer 2000] and Verelst [Verelst 1997] are valuable in addressing the issue that multiple correct conceptual models may exist for a particular application domain, but differ in their evolvability characteristics. Stability was considered as a major contributor towards achieving evolvability.
Approaches for (explicitly) specifying evolving requirements were also addressed in terms of behavioural and structural concerns. As described in section 5.1, the primary issue is the distinction of stable parts from evolvable parts – a decision that needs to be made well in advance of the design phase. Such a separation was also suggested as a solution to counter the problem of design erosion in software architectures – described in section 3.1.3.2. The so-called *externalisation* of component interactions emerged as a primary goal and was discussed in section 4.2, both in terms of a specific domain application and for software architectures in general.

The schema evolution problem was then discussed in section 4.3.1 as a means to cope with change affecting established persistent stores of data and program. The major concerns include the propagation of change and preservation of the semantic integrity of the system in terms of its underlying conceptual specification. Technical solutions dominate the field and often tend to be complex. Schema versioning approaches either present significant storage overheads or are complex in nature. At present, however, the solutions do provide a way of coping with compatibility and consistency concerns – both in terms of dependent application systems and database instances of the schemas.
As far as future work is concerned, novel approaches deserve consideration. In particular, Parsons and Wand [Parsons and Wand, 2000] tackle the more generic and underlying problem of *preferred classification*. They propose an instance-based model (as opposed to class-based) as a vehicle for solving the problems resulting from this. Here, membership of instances to classes is defined purely in terms of the properties that the instance possesses. In particular, users are no longer limited to accessing data through a designer's preferred schema, and schema evolution issues

are also avoided in the sense that classes can now be changed without reference to the underlying population. The approach is unconventional with constraint definition, query capabilities, security restrictions, and general performance aspects being noted as requiring further consideration regarding any implementation of the model. However, these should not detract from the potential benefits that can be realised.

Finally, the problem of evolution is one requiring proactive as well as reactive solutions for any given application domain. Even "best designs" are limited in dealing with unanticipated evolution and require implementation phase paradigms that can facilitate an evolution correctly (semantic integrity), efficiently (minimal disruption of services) and consistently (all affected parts are consistent following the change). Orthogonal persistence and meta-modelling frameworks, such as the MDA, present significant progress in this direction.

While presenting different approaches towards dealing with the evolution problem over the SDLC, it is hoped that this research also indicates that accommodating and facilitating information system evolution requires more emphasis on the conceptual and specification artefacts. This prevents the traditionally steep maintenance overheads that characterise any realisation of requirements change on operational systems. Moreover, the benefits of presenting simpler and less technical development platforms for designers and system procurers are considerable.

CHAPTER 7 - REFERENCES

[Andrade and Fiadeiro, 2001] Andrade L and Fiadeiro J. *Coordination Technologies for Managing Information Systems Evolution,* Proceedings CAISE'01, Dittrich K, Geppert A and Norrie M (Eds), LNCS 2068, Springer-Verlag, 2001, pp. 374-387.

[Asimolar 1998] Bernstein P, Brodie M, Ceri S, DeWitt D, et al. *The Asimolar Report on Database Research,* ACM SIGMOD Record 27(4), December 1998.

[Atkinson and Jordan, 2000] Atkinson M and Jordan M. *A Review of the Rationale and Architectures of PJama: a Durable,Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform,* Sun Microsystems Technical Report, TR-2000-94, June 2000.

[Atkinson and Morrison, 1995] Atkinson M and Morrison R. *Orthogonally Persistent Object Systems,* VLDB Journal 4, 3 (1995), pp. 319-401.

[Atkinson et al, 1996] Atkinson M, Daynes L, Jordan M, Printezis T and Spence S. *An Orthogonally Persistent Java,* ACM SIGMOD Record, December 1996.

[Balko 2000] Balko S. *Adaptive specifications of technical information systems,* Database Schema Evolution and Meta-modelling, Balsters H, de Brock B, Conrad S (Eds), LNCS 2065, Springer-Verlag, 2000, pp. 61-67.

[Balzer 1986] Balzer R and Goldman N. *Principles of good software specification languages,* Appearing in *Software Specification Techniques,* Gehani N and McGettrick (Eds), Addison-Wesley, 1986, pp. 25-39.

[Banker et al, 1993] Banker R, Datar S, Kemerer C and Zweig D. *Software Complexity and Maintenance Costs.* Communications of the ACM, Vol. 36, No. 11, 1993, pp. 81-94.

[Bezivin and Ploquin, 2001] Bezivin J and Ploquin N. *Tooling the MDA framework: a new software maintenance and evolution schema proposal.* Position paper for ECOOSE : OOPSLA 2001 (Workshop on Engineering Complex Object-Oriented Systems for Evolution), 2001. Available from http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml, [Accessed 17-Nov-2002].

[Bird et al, 2000] Bird L, Goodchild A and Halpin T. *Object-Role Modelling and XML-Schema,* Proceedings Conceptual Modelling ER 2000, Laender A, Liddle S, Storey V (Eds.), LNCS 1920, Springer-Verlag 2000, pp. 309-322.

[Boehm 1981] Boehm B, *Software Engineering Economics*, Englewood Cliffs NJ, Prenice Hall, 1981.

[Bowers 1993] Bowers D, *From Data to Database*, 2nd edition, Chapman and Hall, 1993.

[Brooks et al, 2001] Brooks P, Estrella F, Kovacs Z, Le Goff JM, Mathers G and McClatchey R. *Handling System Evolution Through Information Abstraction.* Position paper for ECOOSE : OOPSLA 2001 (Workshop on Engineering Complex Object-Oriented Systems for Evolution), Available from http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml, [Accessed 17-Nov-2002].

[Bry and Eisinger, 2001] Bry F and Eisinger N, *Data Modelling with Markup Languages (DM^2L),* University of Munich, Germany, Available from http://www.pms.informatik.uni-muenchen.de/forschung/datamodeling-markup.html [Accessed 1-Nov-2002].

[Bukhres et al, 1996] Bukhres A, Elmagarmid O,Gherfal F, Liu X, Barker K, Schaller T. *The Integration of Database Systems.* Appearing in *Object-Oriented Multidatabase Systems.* Bukhres A and Elmagarmid O (eds) Prentice-Hall, 1996, pp. 37-56

[Chen 1976] Chen P, *The Entity-Relationship Model – Toward a Unified View of Data,* ACM TODS 1, No 1, March 1976.

[Chen et al, 1995] Chen J, McLeod D and O'Leary D. *Domain-Knowledge-Guided schema evolution for accounting database systems.* Expert Systems with Applications, Vol 9, No 4, 1995, pp. 491-501.

[Connor et al, 1994] Connor R, Cutts Q, Kirby G and Morrison R. *Using persistence technology to control schema evolution.* Proceedings 9th ACM Symposium on Applied Computing, Phoenix, Arizona, March 1994, pp. 441-446

[Conrad et al, 1998] Conrad S, Ramos J, Saake G and Sernadas C. *Evolving Logical Specifications in Information Systems.* In Chomicki J and Saake G (eds), *Logics for Databases and Information Systems*, Kluwer Academic Publishers, Boston, 1998, Chapter 7, pp. 199-228.

[Costello and Schneider, 2000] Costello R and Schneider J, *Challenge of XML Schemas – Schema Evolution*, MITRE Corp., May 29, 2000. Available from http://www.xfront.org/evolvableschemas.html, [Accessed 24-August-2002].

[Date 1990] Date C, *An Introduction to Database Systems*, Volume 1, 5th Edition, Addison-Wesley, 1990.

[Dmitriev, 2000] Dmitriev M, *Class and Data Evolution Support in the PJama Persistent Platform*. Technical Report TR-2000-57, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK, November 2000.

[Falkenberg et al, 1992] Falkenberg E, Oei J and Proper H. *Evolving Information Systems: Beyond Temporal Information Systems*. In Tjoa A and Ramos I (Eds). *Proceedings of the Data Base and Expert System Applications Conference (DEXA 92)*, Valencia, Spain, September 1992, pp. 282-287.

[Franconi et al, 2000] Franconi E, Grandi F and Mandreoli F. *A semantic approach for schema evolution and versioning in object-oriented databases.* Lecture Notes in Artificial Intelligence 1861, Springer-Verlag, Berlin 2000, pp. 1048-1062.

[Gabriel et al, 1993] Gabriel R, Bobrow D and White J. *CLOS in Context – The Shape of the Design Space.* In Object-Oriented Programming – the CLOS perspective, Chapter 2, MIT Press 1993, pp. 29-61.

[Ghose 1999] Ghose A, *Managing Requirements Evolution: Formal support for functional and non-functional requirements*, Proceedings of the 1999 International Workshop on Principles of Software Evolution, Fukuoka, Japan, 1999, pp. 118-124.

[Halpin 1995] Halpin, T. *Conceptual Schema & Relational Database Design*, Second Edition, Prentice-Hall, 1995

[Hay 1999] Hay D, *A Comparison of Data Modelling Techniques*, Essential Strategies Inc., October 1999. Available from http://www.essentialstrategies.com, [Accessed 30-July-2002].

[Hofmann et al, 1996] Hofmann C, Horn E, Keller W, Renzel K, Schmidt M. *The Field of Software Architecture*. Technical Report TUM - I9641, Technische Universität München, 1996

[Hopkins 2000] Hopkins J, *Component Primer,* Communications of the ACM 43(10), 2000, pp 27-30.

[Jungclaus et al, 1991] Jungclaus R, Saake G, Hartmann T and Sernadas C. *Object-oriented specification of information systems: The TROLL language*. Informatik-Bericht 91-04, Technishe Universität Braumschweig, 1991.

[Kim and Carrington ,2000] Kim S and Carrington D, *A Formal Mapping between UML Models and Object-Z Specifications,* appearing in Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29 - September 2, 2000, Proceedings, LCNS, 1878 Springer 2000

[Kirby et al, 1997] Kirby G, Morrison R, Connor R Zdonik S. *Evolving Database Systems: A Persistent View.* University of St. Andrews Technical Report CS/97/5 (1997)

[Lerner 2000] Lerner B. *A model for compound type changes encountered in schema evolution.* ACM Transactions on Database Systems, Vol 25, No 1, March 2000, pp. 83-127

[Lientz and Swanson, 1978] Lientz, B and Swanson E, *Characteristics of application software maintenance*, Communications of the ACM, 21, 6 (June 1978), pp 466-481.

[Lientz and Swanson, 1980] Lientz B and Swanson E, *Software Maintenance Management*, Reading MA, Addison-Wesley, 1980.

[Lientz and Swanson, 1981] Lientz B and Swanson E. *Problems in Application Software Maintenance.*
Communications of the ACM, Vol. 24, No. 11, 1981, pp. 763–769.

[Liu 1998] Liu L. *EVOLVE: Adaptive Specification Techniques for Object-oriented Software Evolution.* In: the 31st Hawaii International Conference on Systen Sciences (HICSS-31) -- Modelling Technologies and Intelligent Systems Track, January 6-9, 1998

[Mani et al, 2001] Mani M, Lee D and Muntz R. *Semantic Data Modelling using XML Schemas.* Conceptual Modelling – ER 2001, Kunii H, Jajodia S, Solvberg A (Eds.), LNCS 2224, Springer-Verlag 2001, pp 149-163.

[Mens et al, 1999] Mens K, Mens T, Wouters B and Wuyts R. *Managing unanticipated evolution of software architectures*, in Proceedings Object-Oriented Technology, ECOOP'99, Moreira A, Demeyer S (Eds), 1999, pp. 75-76. Also available at http://citeseer.nj.nec.com/mens99managing.html. Accessed [17-Nov-2002].

[OMG-MOF 1997] OMG/MOF *Meta Object Facility (MOF) Specification*. OMG Document AD/97-08-14. September 1997.

[OMG-MOF, 2000] OMG, *MetaObject Facility (MOF) Specification*, Version 1.3, March 2000. Available at http://www.omg.org/technology/documents/formal/mof.htm [Accessed 17-September-2002].

[OMG-Soley, 2000] OMG: Soley R and OMG Staff Strategy Group, *Model Driven Architecture*, White paper, Draft 3.2, November 2000. Available at http://www.omg.org. [Accessed 17-September-2002].

[Parsons and Wand, 2000] Parsons J and Wand Y. *Emancipating instances from the tyranny of classes in information modelling*. ACM Transactions on Database Systems, Vol 25(2), June 2000, pp 228-268

[Polack, 1992] Polack F, *Integrating formal notations and system analysis: using entity relationship diagrams*, Software Engineering Journal, September 1992, pp 363-371.

[Pons and Keller, 1997] Pons A and Keller R. *Schema evolution in object databases by catalogs*. Proceedings of the 1997 Intl. Database Engineering and Applications Symposium (IDEAS'97), 1997 IEEE.

[Poole, 2001] Poole J, *Model-Driven Architecture : Vision, Standards and Emerging Technologies*, Position paper submitted to ECOOP 2001 : Workshop on Metamodeling and Adaptive Object Models, Hyperion Solutions Corporation, April 2001. Available at www.cwmforum.org/Model-Driven%20Architecture.pdf , [Accessed 17-September-2002]

[Potter et al, 1996] Potter B, Sinclair J and Till D, *An Introduction to Formal Specification and Z*, Second Edition, Prentice Hall International Series in Computer Science, 1996.

[Ra and Rudensteiner, 1997] Ra Y and Rudensteiner E. *A Transparent Schema-Evolution System based on Object-Oriented View Technology*. IEEE Transactions on Knowledge and Data Engineering, Vol 9. No 4, July/August 1997.

[Riebisch and Philippow, 2001] Riebisch M and Philippow I, *Evolution of Product Lines using Traceability*, Position paper for ECOOSE : OOPSLA 2001 (Workshop on Engineering Complex Object-Oriented Systems for Evolution), 2001. Available from http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml, [Accessed 17-November-2002].

[Roddick 1995] Roddick J. *A survey of schema-versioning issues for database systems*. Information and Software Technology 1995. 37 (7) pp. 383-393

[Saake et al, 2000] Saake G, Türker C, and Conrad S, *Evolving Objects: Conceptual Description of Adaptive Information Systems,* Database Schema Evolution and Meta-modelling, Balsters H, de Brock B, Conrad S (eds), LNCS 2065, Springer-Verlag, 2000, pp. 163-181

[Shaw and Garlan, 1996] Shaw M and Garlan D, *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, N.J., Prentice Hall, 1996

[Simsion and Shanks, 1993] Simsion G and Shanks G. *Choosing Entity Types – a study of 51 data modellers.* Technical Report 17/93, Monash University, Melbourne, Australia, 1993.

[Sjoberg 1993] Sjoberg D, *Managing Change in Information Systems : Technological Challenges.* Available at http://citeseer.nj.nec.com/118144.html. Accessed [17-November-2002].

[Sommerville 1992] Sommerville I, *Software Engineering*, 4th edition, Addison-Wesley, 1993

[Soni 1995] Soni D, Nord R, and Hofmeister C. *Software Architecture in Industrial Applications,* Proceedings of the 17th International Conference on Software Engineering, Seattle, WA. Available at http://citeseer.nj.nec.com/soni95software.html. [Accessed 17-November-2002].

[Stemple et al, 1992] Stemple D, Stanton R, Sheard T, Philbrow P, Morrison R, et al. *Type-Safe Linguistic Reflection: A Generator Technology.* ESPIRIT BRA Project 3070 FIDE Technical Report FIDE/92/94, 1992.

[Tanaka et al, 2000] Tanaka K, Ghandeharizadeh S and Kambayashi Y, *Information Organisation and Databases*, appearing in *Information Organisation and Databases – Foundations of Data Organisation*, Ghandeharizadeh S and Kambayashi Y (eds), Kluwer Academic Publishers, 2000.

[Van Gurp and Bosch, 2001] Van Gurp J and Bosch J, *Design Erosion: Problems and Causes,* Journal of Systems and Software 61 (2002), pp 105-119.

[Verelst 1997] Verelst J, *Factors in Conceptual Modelling Influencing Evolvability of Information Systems.* In: Proceedings of the Doctoral Consortium of the 3rd IEEE International Symposium on Requirements Engineering [1997: Annapolis, Calif.], 1997, pp. 93-100

[Verelst 1998] Verelst J. *The influence of domain-specific abstraction on evolvability of software architectures for information systems.* In: Object-oriented technology: ECOOP'98 Workshop reader / Demeyer S, et al.. (Eds), Berlin, Springer, 1998, pp. 56-57

[W3C Malhotra 1999] World-Wide Web Consortium (W3C), *XML Schema Requirements,* Malhotra A and Maloney M (eds), W3C Note 15 February 1999. Available at http://www.w3.org/TR/1999/NOTE-xml-schema-req-19990215. [Accessed 17-September-2002].

[Wedemeijer 1999] Wedemeijer L. *Design the flexibility, maintain the stability of conceptual schemas.* Lecture Notes in Computer Science 1626, 1999, pp461-471.

[Wedemeijer 2000] Wedemeijer L, *Defining metrics for conceptual schema evolution,* 9th International Workshop on Foundations of Models and Languages for Data and Objects, Germany, Sept 2000, Springer-Verlag series LCNS 2065, pp. 220-244.

[Weiderhold 1995] Weiderhold G, *Modelling and System Maintenance.* Appearing in *OOER'95: Object-Oriented and Entity Relationship Modelling*; Proceedings, ER-OO conference, Papazoglou, M (ed.) LNCS, 1021, December 1995, pp 1-20.

[Wing 1990] Wing J M, *A specifier's introduction to formal methods,* Computer, 23(9), 1990, pp. 8-22.

[Wu and Han, 2002] Wu J and Han J, *xmlTRAM+: Using XML Technology to Manage Software Requirements and Architectures,* The Eighth Australian World Wide Web Conference, July 2002, Available at http://ausweb.scu.edu.au/aw02/papers/refereed/wu/paper.html [Accessed 1-November-2002].

[Zaniolo et al, 1997] Zaniolo C, Ceri S, Faloutsos C, Snodgrass R, Subrahmanian Vand Zicari R. *Advanced Database Systems.* Morgan Kaufmann, 1997

GLOSSARY OF ABBREVIATIONS AND ACRONYMS

.NET	Pronounced "dot net". A Microsoft operating system platform primarily designed to facilitate development of interoperable Web applications. This incorporates applications, in addition to a suite of tools and services.
AI	Artificial Intelligence: Broadly, the study of how to make computers perform tasks that are currently better performed by humans. In the context of this research, reasoning and natural language understanding are relevant.
ANSI/SPARC	American National Standards Institute/Systems Planning and Requirements Committee. Used to refer to the three-level (internal, external, conceptual) architecture to model database systems.
AOM	Adaptive Object Model: Implementations employing such technology provide dynamic system behaviour based on the runtime interpretation of meta-models.
C++	An object-oriented programming language.
CAD	Computer Aided Design: CAD software is used by engineers, architects, etc., to create precision drawings or technical illustrations in 2-dimensional or 3-dimensional form. The management of the software library of design components and the relationships between design components are relevant to this research in the database schema sense. Also see CAM.
CAM	Computer Aided Manufacturing: Computer-aided control of the manufacturing process for a product. CAD/CAM systems allow engineers to design a product and control its manufacturing process. The required information system management facilities for such environments are relevant to this research.
CASE	Computer Aided Software Engineering: Automated support for software engineering.
CORBA	Common Object Request Broker Architecture: An architecture that enables program objects to communicate with one another regardless of what programming language they were written in or what operating system they're running on. CORBA is an OMG standard.

CWM	Common Warehouse Meta-model: A MOF-compliant meta-model for representing both the business and technical metadata that's most often found in data warehousing and business analysis domains. Provides support for the ability to model legacy and non-legacy data resources, including relational databases, record-oriented databases, and XML- and object-based data resources.
DAG	Directed Acyclic Graph: In this research, used to refer to an object-oriented schema where the relationships between classes must be such that they resemble a directed graph without cycles.
DBMS	Database Management System: A collection of programs that enable the storage, modification, and extraction of information from a database.
DDL	Data Definition Language: Language supporting the definition or declaration of database objects.
DFD	Data Flow Diagram: These graphically illustrate how input data is transformed to output results through a sequence of functional transformations.
DTD	Document Type Definition: A DTD states what tags and attributes are used to describe content in an SGML document, where each tag is allowed, and which tags can appear within other tags.
EBNF	Extended Backus-Naur Form: BNF (Backus-Naur Form) is used to formally define the grammar of a language. EBNF is a variation on the basic BNF meta-syntax and includes additional constructs.
e-commerce	Electronic Commerce: Conducting business on-line (typically over the internet).
EJB	Enterprise Java Beans: A Java application programming interface that defines a component architecture for multi-tier client-server systems. The EJB component model simplifies the development of middleware applications by providing automatic support for services such as transactions, security, database connectivity, and more.
ER	Entity-Relationship. In the sense of Chen's Entity-Relationship approach described in section 2.2.1.
EVOLVE	A object-oriented notation described in [Liu, 1998] for specifying structural and behavioural requirements.

HTML	Hypertext Markup Language: The authoring language used to create documents on the World Wide Web. HTML is similar to SGML, although it is not a strict subset. HTML defines the structure and layout of a Web document by using a variety of tags and attributes.
Java	An object-oriented programming language similar to C++. Designed to be executable on different platforms. Developed by Sun Microsystems.
Java RMI	Java Remote Method Invocation: RMI is the Java version of what is generally known as a remote procedure call (RPC), but with the ability to pass one or more objects along with the request.
JDBC	Java Database Connectivity: A programming interface that enables Java programs to interact with SQL-compliant databases.
JVM	Java Virtual Machine: An abstract computing machine, or virtual machine. JVM is a platform-independent execution environment that converts compiled Java code into machine language for execution.
MDA	Model-Driven Architecture: An OMG initiative for system specification and interoperability based on the use of formal models.
MOF	Meta Object Facility: An OMG standard defining a common, abstract language for the specification of meta-models, such as CWM. MOF is an example of a meta-meta model (or model of the meta-model).
NIAM	Nijssen's Information Analysis Methodology: A predecessor of the ORM approach to conceptual modelling. Was later generalised to "Natural language Information Analysis Method". Also see ORM.
OMG	Object Management Group: A consortium providing a common framework for developing applications using object-oriented programming techniques.
OMT	Object Modelling Technique: Uses DFDs, hybrid E-R diagrams, and statecharts to model software requirements using object-oriented concepts. The OMT notations are only partially formal.
ORM	Object Role Modelling: A conceptual modelling technique developed by Halpin. It describes objects and their relationships, as well as domain constraints in a formal (graphical) notation. It involves a step-by-step design procedure based on verbalisation in natural language. Further detail can be found in [Halpin 1995].

PAS	Persistent Application System: Systems where the application as such outlives its individual components and even its implementation technology.
PIM	Platform-Independent Model: The basis of an MDA-based application. It is defined in terms of UML, allowing an application model to be constructed, viewed, developed and manipulated in a standard way at analysis and design time.
PJama	A research project in progess at the University of Glasgow, Scotland in conjunction with Sun Microsystems. It is aimed as a vehicle for implementing and testing an orthogonal persistent version of the Java programming language. Orthogonal persistence is described in section 4.3.3.1.
PSM	Platform-Specific Model: The platform specific interpretation of a PIM. It contains the same information as a fully-coded application, but is expressed in UML instead if code and associated files.
SDLC	Software Development Life Cycle: The process of developing information systems through requirements analysis, specification, design, implementation and maintenance.
SGML	Standard Generalized Markup Language. A system for organising and tagging elements of a document. SGML was developed and standardised by the International Organization for Standards (ISO) in 1986.
SOAP	Simple Object Access Protocol. This provides a way for applications to communicate with each other over the internet independent of platform. It is an XML-based protocol that is designed to exchange structured and typed information on the Web. It consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses.
SQL	Structured Query Language: Standardised declarative language for formulating relational operations (i.e. operations that define and manipulate data in relational form).
TESS	Type Evolution Software System: An acronym for Lerner's [Lerner 2000] software tool for dealing with compound type changes.
TROLL	A language for the object-oriented specification of information systems. It is designed to describe the Universe of Discourse (UoD) as a system of concurrently existing and interacting objects. See [Jungclaus, et al, 1991].

TSE	Transparent Schema-Evolution System: An acronym for Ra and Rudensteiner's [Ra and Rudensteiner, 1997] tool for managing schema versioning.
UIMS	User Interface Management System: Allows a programmer to "connect" the behaviour at the user interface with the underlying functionality of a system.
UML	Unified Modelling Language: An OMG notation used for representing the structure of data in object-oriented systems. It also adds the ability to describe the behaviour of each object class/entity.
UoD	Universe of Discourse: Used to refer to the real world domain underlying an application.
W3C	World Wide Web Consortium: An international consortium of companies involved with the Internet and the Web. The organisation's purpose is to develop open standards so that the Web evolves in a single direction rather than being splintered among competing factions.
XMI	XML Metadata Interchange: An OMG standard that maps the MOF to XML. XMI effectively defines how XML tags are used to represent serialised MOF-compliant meta-models in XML.
XML	Extensible Markup Language. A specification developed by the W3C. XML is a subset of SGML, designed especially for Web documents. Designers are able to create their own customised tags (a command inserted into a document, specifying the format of a document or part thereof) enabling the definition, transmission, validation, and interpretation of data between applications and between organisations.
Z	A model-based formal software specification language. It uses mathematical concepts and notation, including set theory, to build models of systems.