Understanding Object-Relational Mapping: A Framework Based Approach

Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh
Department of Computing
The Open University
Milton Keynes, UK
e-mail: cji26@student.open.ac.uk, (D.S.Bowers, M.A.Newton, K.G.Waugh)@open.ac.uk

Abstract - **Object and relational technologies are grounded in different paradigms. Each technology mandates that those who use it take a particular view of a universe of discourse. Incompatibilities between these views manifest as problems of an object-relational impedance mismatch. In a previous paper we proposed a new conceptual framework for the problem space of object-relational impedance mismatch and consequently distinguished four kinds of impedance mismatch. Here we show how that framework provides a mechanism to explore issues of fidelity, integrity and completeness in the design and implementation of an existing object-relational mapping strategy. We propose a four-stage process for understanding a strategy. Using our process we show how our framework helps to identify new issues, understand cause and effect, and provide a means to address issues at the most appropriate level of abstraction. Our conclusions reflect on the use of both the framework and the process. The information arising from the use of our framework will benefit standards bodies, tool vendors, designers and programmers, as it will allow them to address problems of an object-relational impedance mismatch in the most appropriate way.**

Keywords: Object-Relational; Impedance Mismatch; ORM; Framework

I. INTRODUCTION

In [1] we provide a new framework for understanding the problem space of object-relational impedance mismatch. If we address the root cause of an object-relational impedance mismatch problem rather than the symptoms as we do today, we will reduce the cost of software development by avoiding the quagmire described by Neward [2] and discourage others (such as [3]) from reinventing solutions.

A paradigm is a particular way of viewing a universe of discourse. Each paradigm comes with its own particular abstractions, organising principles and prejudices. There are a number of different paradigms in computing. Each paradigm has influence on both the process and artefacts of software design and development.

The combination of technologies based on different paradigms presents a set of problems for those responsible for the design and implementation of an application. We refer to each such problem as an impedance mismatch problem. People are inventive and proponents of one paradigm may believe that they have solved an impedance mismatch problem. Such a solution will typically involve using a subset of concepts from one paradigm to represent a concept in the other. It then becomes received wisdom within a community that there is a solution to a problem and that all those concerned understand the solution.

The relational paradigm has proven popular in the development of databases whilst at the same time the object paradigm has underpinned a number of programming languages and software development methods. The popularity of technologies that embody different paradigms in these two separate but essential aspects of software development means that inevitably they will be used together. Differences in abstraction, focus, language etc. between paradigms however leads to problems when these technologies are combined in a single application.

An object-relational application combines artefacts from both object and relational paradigms. Essentially an object-relational application is one in which a program written using an object-oriented language uses a relational database for storage and retrieval. A programmer must address object-relational impedance mismatch ("impedance mismatch") problems during the production of an object-relational application. For some authors [4] however there is no impedance mismatch. This is true for those developing an entire application using a single programming language such as Visual Basic, C++, Java or SQL-92[1] ("SQL") because each language is based on a single paradigm. Those who have to combine object and relational technologies and must work across paradigms have a different experience [5], [2]. The received wisdom is that these impedance mismatch problems are both well understood and resolved by current solutions based on SQL. For each such impedance mismatch problem however there is a choice of solution. We refer to each such solution as an Object-Relational Mapping (ORM). Each ORM strategy addresses problems of an impedance mismatch in a different way. We seek to understand the most appropriate way to address a problem.

During the development of an object-relational application based on SQL-92, the resolution of impedance mismatch problems involves many different roles and takes time and effort to achieve [2]. Neward [2] labelled the problem of impedance mismatch "the Vietnam of Computer Science" because initial quick wins based on the received wisdom are rapidly replaced by a quagmire of

[1] This work is presented in the context of mapping from an OOPL to SQL-92, which does not include Object Relational (OR) extensions. Future work will analyse the effectiveness of the OR extensions to SQL in addressing ORIM problems.

issues. Keller in [6] claims that twenty five to fifty percent of object-relational application code is concerned with problems of an impedance mismatch. The popularity of object and relational technologies, the plethora of solutions and technologies for the resolution of an impedance mismatch, and the existence of guidelines [7] and metrics [8] for selecting a strategy also suggest that problems of an impedance mismatch are neither uncommon nor trivial.

In this paper we propose a four-stage process for understanding a strategy. Using our process we show how our framework helps to identify new issues with a strategy, understand cause and effect, and provide a means to address those issues at the most appropriate level of abstraction.

The rest of the paper is structured as follows. Section II presents the impedance mismatch problem space; Section III presents an analysis of current approaches to ORM; Section IV presents our framework; Section V relates our framework to ORM strategies; Section VI presents our process for using the framework; Section VII provides a worked example; and Section VIII presents our conclusions and future work.

II. Problems of an Object-Relational Impedance Mismatch

In the context of object-relational application development, one objective of an ORM strategy is to isolate a programmer using an object-oriented programming language (OOPL) from the need to understand the SQL language, the schema of an SQL database, and its implied semantics. A programmer need not focus on how to store an object but on what to store and when to store it, and what to retrieve and when to retrieve it.

Such a strategy is typical of ORM products such as Hibernate [9] and Oracle TopLink. They provide a programmer with a virtual object database, presenting data in a relational database as if it were a collection of objects. However, ORM does not isolate a relational database from an object-oriented program. The design of a relational database must address issues such as data redundancy, data integrity, data volumes, access control, concurrency, performance and auditing. Impedance mismatch problems occur when these requirements are at odds with the design of an object-oriented program. These problems have been described by writers such as [2] and [5]. In Table I we have catalogued the issues emerging from their work as problems of an object-relational impedance mismatch (ORIM).

TABLE I. Problems of an Object-Relational Impedance Mismatch

Problem	Description of the problem and typical questions raised
Structure	A class has both an arbitrary structure and an arbitrary semantics defined through methods. A class may also be part of a class hierarchy. SQL-92 does not provide an analogy for a class hierarchy or support repeating groups within a column. How then do we best represent the structure of a class using SQL?
Instance	To conform to relational theory, a row is a statement of truth about some universe of discourse, but an object is an instance of a class and may have an arbitrary structure. How does a row correspond to an object and where is the canonical copy of state located? Essentially, how much of an object must we maintain in a database?
Encapsu-lation	The state of an object is accessed via methods. The state of a row has no such protection and may be modified by other applications. How do we ensure consistency of state between an object and a row?
Identity	An object has an identity independent of its state. This in-memory identity will be different between two executions of a program. Within the same execution, two objects with the same state are different if they have a different identity. The primary key of a row is part of the state of that row. How do we uniquely identify a collection of data values across both object and relational representations?
Process-ing Model	An object model is a network of interacting discrete objects and access is based on navigation. The relational model is declarative and access is set-based. The object and relational models represent references in different directions. A transaction may not require all the data about an object. How do we represent in, maintain in, and retrieve from a database a sufficient set of in-memory objects?
Owner-ship	A class model is owned by a programming team, a relational schema is ultimately owned by a database team, it may hold legacy data and may also be used by other applications. When things change how do we maintain the necessary correspondence between a class model and a database schema?

III. Object-Relational Mapping

In the literature and in practice we find many examples of ORM ([3],[10],[11],[12],[13], and [14]). Essentially an ORM strategy is how we address each problem of an impedance mismatch but in research and practice the term ORM is used to refer to a number of different things: for Fussell [11] it is a transformation process; for others ORM is something defined in the configuration of a mapping tool such as Hibernate [9]; whilst to others it is a pattern [13] or canonical mapping [14] used as the basis for a design transformation.

Practitioners recognise ORM as both a process and a mechanism ([5], p225) by which an impedance mismatch is addressed. As a process, ORM is the act of determining how objects and their relationships are made persistent in a relational database: in essence the selection of one or more patterns [13]. These patterns are based on the assumption that an object model is used as the basis for a database schema and that schema confirms to SQL-92. They do not help with the development of an object-based application that uses a legacy relational database.

As a mechanism, ORM forms the definition of correspondence necessary for the successful implementation of a particular pattern as one or more mappings. A mapping relates two representations in different implementation languages. In order to address an

impedance mismatch, this mapping is codified as one or more transformations within some part of an application. For the developer of an object-oriented application that must use a relational database for persistence ORM may be all of these, impedance mismatch is also a fact of life [13]. We observe from this variety that there is no consensus on a single strategy for ORM and by implication how we address impedance mismatch. Each strategy addresses a different aspect of impedance mismatch. Some strategies focus on equivalence between a class and a table [13] (what to map) whilst others propose a unified query language [15] (how to map) or software architecture [11] (where to map). It is not clear whether any of these strategies address the root case of an impedance mismatch problem or whether they make do with the facilities available. Evidently when these writers use the term "impedance mismatch" they are not talking about the same thing. We require some form of organising principle which goes beyond received wisdom to facilitate an understanding and comparison of strategies, and which also recognises an essential aspect of the problem: the different levels of abstraction.

It is evident from Table I that impedance mismatch is not a single, well-defined problem. The different interpretations of ORM also indicate there is no single, well-defined solution. If we are to understand impedance mismatch we must understand the nature of these different problems and how they are addressed by different approaches to ORM. At a detailed level this understanding provides the motivation for our conceptual framework and classification.

IV. A CONCEPTUAL FRAMEWORK OF OBJECT-RELATIONAL MAPPING

In this section we consider how we might organise the different views of ORM. Other models such as [11] and [16] focus on client/server software architecture. Essentially they help inform where one might perform a mapping. Hohenstein [12] considers programming language issues and helps to inform a C++ programmer how to perform a mapping.

The rationale and the motivation for our framework and classification were established in [1]. Our framework comprises four levels of abstraction common to both object and relational technologies. The classification allows us to organise the different issues at each level. These levels (Table II) allow us to understand why we are performing a mapping and allow us to identify the root cause of a problem. Object and relational silos span all four levels. Within each level there are therefore both object and relational contributions. We summarise our framework in Figure 1.

The levels are labelled using terms that may themselves have alternative interpretations and therefore require clarification. A paradigm is one particular way of viewing a universe of discourse ([17], pA-6). A language is used to produce an abstract description of a universe of discourse. We consider a concept to be some identifiable collection of things from a universe of discourse. A schema is a description (representation) of some concept from a universe of discourse, expressed using a particular language. We consider program source code the schema for an executing program just as an SQL script is the schema for a relational database. Finally an instance is data about some thing from the universe of discourse set within a particular schema.

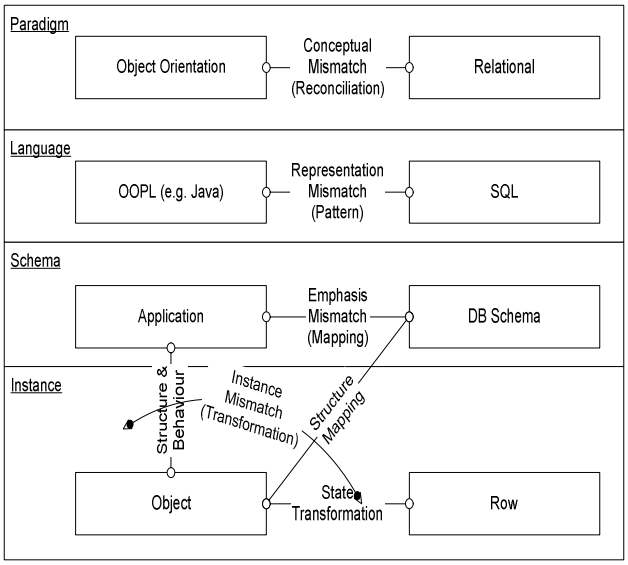

Figure 1. Our conceptual framework

The relationship between the levels of our conceptual framework is one of context. A paradigm sets the context for the semantics of a language. A language provides data and processing structures for describing the semantics of a universe of discourse in the form of a schema. There are many possible schemata. A schema sets the structure into which data about some thing from a universe of discourse must fit. Conversely a schema sets constraints on what it is we can represent about some thing from a universe of discourse.

TABLE II. OUR CONCEPTUAL FRAMEWORK OF OBJECT-RELATIONAL MAPPING

Level	ORM is concerned with…
Paradigm	Issues relating to the incompatibilities between the two different views of a concept from a universe of discourse: one as a network of interacting objects and the other as a set of relations.
Language	Issues relating to the incompatibility of data structures between object and relational based languages. ([14], p182) refer to this as a canonical mapping. In this paper we will use the term pattern in the context of [13] as an outline description of a solution.
Schema	Issues relating to the maintenance of two representations of a particular concept described in different languages.
Instance	Issues relating to the storage and retrieval of an object in the context of an object-relational application.

Contextualisation has implications for choices made during the development of an object-relational application. A development language brings with it not only an implicit choice of paradigm but also a set of structures and patterns that may be used ([5], [12], and [13]). The maintenance of a legacy application may dictate the use of a particular language. Choices made during the design of an object model and an SQL schema dictate the content of a mapping schema. During the development of an object-relational application, the teams responsible for program and database schema development will make their own choices based on their own agenda.

A programmer has many technologies and algorithms from which she may choose in order to implement a transformation: for Java alone there is a choice of JDBC, Hibernate, JDO and Oracle TopLink to name a few. All levels of our conceptual framework have influence on the work a programmer must do in order to resolve an impedance mismatch. When we use the term ORM we must recognise that an impedance mismatch problem has its source at any of these levels, and understand how and at what level(s) a problem is best addressed.

Analysis of ORM strategies in the literature such as [13] have focussed on consequences in implementation rather than understanding the underlying issues with a strategy. Our framework provides an organising mechanism that allows us to explore issues in the design and implementation of existing and new ORM strategy choices. Achieving an understanding of the underlying issues is an important contribution of our framework.

V. ORM Strategies and our Four Level Framework

Our framework provides a new way to think about the problem of impedance mismatch and how we go about resolving it. Each level provides a different way of thinking about an ORM strategy. In this section we explore the relationship between problems of an impedance mismatch and the layers of our framework. For each strategy we provide illustrations from the literature.

A. Paradigm

An ORM strategy at the paradigm level involves the reconciliation of different perspectives of a universe provided by the object and relational paradigms. Different aspects of an object-relational application are grounded in each paradigm. Typically the object paradigm influences program design and the relational paradigm database design. ORM in this context is the act of bridging the differences between these two paradigms. This is the essence of the impedance mismatch problem. It is therefore important to understand the nature of these differences.

There is no consensus of terminology. Each paradigm uses a different set of building blocks to describe a universe of discourse. Although there is no single agreed definition of an object-based representation (the UML is one attempt but there are others [18]), such a representation will typically include concepts such as class, subclass, object, attribute, and association. There is however a single definition of what constitutes a relational representation [19]. A relational representation of the same universe will include concepts such as relation, tuple and domain.

There is some correspondence between the building blocks. The relational paradigm does not prescribe the domains that may be used. Neither does an object paradigm prescribe the objects that may be used. A relation represents an assertion (a predicate) about a universe of discourse involving one or more domains and a tuple of a relation is formally a statement of truth about that universe. There is however no equivalent representation in the object paradigm. An object is not a representation of a statement of truth about a universe of discourse. Furthermore an object has identity and encapsulates its state whereas a tuple does not. A class defines the allowable attributes and behaviour of an object but its definition is not based on predicate logic. Whereas the relational model is concerned with statements of truth, an object has arbitrary semantics. The behaviour of an object is defined using methods and a valid state is defined using a constraint. In this respect a tuple may be considered inert in so far as it has no intrinsic behaviour. Instead a tuple may be the target of a relational operator such as project, restrict or union. A lack of correspondence between two perspectives on a universe of discourse materialises as an impedance mismatch. We label this kind of impedance mismatch a *conceptual mismatch* and it is addressed using an ORM reconciliation strategy.

A reconciliation strategy must address differences in perspective, terminology and semantics. The designer of an object representation and the designer of a relational representation view and describe aspects of a universe of discourse in different ways. The designer of an object-relational application must identify correspondence and reconcile differences between these two perspectives.

One example of the reconciliation of object and relational semantics is provided by Date [20]. He emphasises that relational theory is not at odds with the ideas of object-orientation. Just as the semantics of a class are arbitrary, the relational model does not prescribe the data types that may be defined. There is therefore scope for addressing a conceptual mismatch.

B. Language

An ORM strategy at the language level is concerned with identifying general patterns of correspondence between the data structures available in an OOPL such as Java, and those structures available in SQL.

Each language reflects the paradigm on which it is based. The outline structure of a Java program is a collection of classes. A class may be viewed as a template for the creation of an object at run-time. An SQL schema is a description of a collection of tables. A table corresponds to a relation. Whereas, formally, a tuple is a statement of truth, the semantics of a row are less strict. A row represents data about some thing from a universe of discourse. Each row corresponds to a tuple.

A significant difference between Java and SQL is the extensibility of their type systems. Whereas a class is an essential part of the extensibility of the Java type system there is no equivalent extensibility in the SQL type system. Implementing a representation of a relation in an OOPL [21] or an SQL like syntax within an OOPL [15] may move the primary focus of ORM activities to the schema level, but it does not address this extensibility issue or remove the need for an ORM strategy.

Generally a class is part of the type system of an object-oriented program. Using SQL a class is something that may be represented, it is not an extension of the type system and also not a first-class citizen. This is the essence of a structure problem and there exist a number of patterns to help resolve this [13]. Aspects of an object-oriented design such as a class and an object fit into a *representation* that must be described using the existing features of SQL. A column is a scalar value and cannot adopt the type of a class represented in such a way. This representation is limited as it may only be used to store the state and not the behaviour of an object. In an object-oriented application at run-time an object has a unique identity independent of its state. The value of a primary key is part of the state of a row. This is the essence of the identity problem. The mismatch between two descriptions of a concept materialises as an impedance mismatch. We label this kind of impedance mismatch a *representation mismatch* and it is addressed using an ORM pattern strategy.

A pattern provides a way to describe correspondence between data structures. SQL provides an approximation of the data structures mandated by the relational paradigm, just as Java provides an approximation of those mandated by the object paradigm. The syntax and grammar for SQL is defined by standard and is implemented in vendor-specific languages such as Oracle and SQLServer. None of these languages is a pure implementation of SQL but nevertheless may be classified as a relational language. In practice we must address not only differences between languages as defined by their respective standards but also differences between vendor implementations. A pattern strategy must provide one or more patterns (such as [13]) that address issues of structure and identity.

C. Schema

An ORM strategy at the schema level will produce a mapping between two representations of a concept. Our emphasis here is on design issues. The description of a concept within an object-relational application will involve at least two schemas: one based on class and the other based on table. These two representations of a concept are different not just because they are phrased in a different language, but because the purpose of those designing a class model is different from the purpose of those designing an SQL schema. Whilst those designing a class will focus on the cohesive representation of a network of interacting objects, the focus of those designing a SQL schema is typically data volume, data integrity, and notably the removal of redundant data. A UML class model may only be familiar to one part of a development team: the programmers. Database designers will conceive a different solution based on tables that may not have a one-to-one correspondence to that class model. This difference of focus is the essence of the ownership problem and produces a kind of impedance mismatch that we label an *emphasis mismatch*. An emphasis mismatch is addressed using an ORM mapping strategy.

A mapping strategy is concerned with correspondence between two different descriptions of a concept. In order to address the ownership problem, this correspondence must be documented, published and implemented. Although the detail is application specific, a mapping strategy will generally provide a mechanism for identifying, documenting, and implementing the correspondence of structure and identity between specific entries in a class model and entries in an SQL schema. Hibernate [9] uses XML whilst [22] make use of meta-data stored in SQL tables. This information forms an important part of the design of an object-relational application.

D. Instance

One issue that lies at the heart of ORM practice is the treatment of an object. The problem is that an object is conceptualised as an atomic unit when in practice it has a number of subdivisions. A Java object has subdivisions of structure, state and behaviour. The schema and instance levels of our conceptual framework show how these subdivisions are fragmented (Figure 2). The structure of an object is defined both in a class and an SQL schema (the ownership problem), the behaviour of an object is defined in a class and a valid state of an object must be maintained and enforced both in-memory and across one or more rows in one or more tables, giving rise to encapsulation and identity problems.

In practice fragmentation is addressed using a transformation but there are problems. Data about an object may not transform cleanly to a row of a table or an individual slot ([20],p3) (the instance problem). The structure of an object may not transform to a single table (the structure problem). The SQL-92 standard does not support the behavioural aspects of an object and so the behaviour of an object must be implemented within a Java class at the schema level. The later introduction of persistent stored modules in SQL provided an opportunity for the fragmentation of behaviour. At run-time it may not be necessary to retrieve all the data about an object for a user to complete a transaction. This combined with fragmentation of the universe of objects required to complete a transaction, is the essence of the processing model problem and provides another driver for ORM transformation activities. Fragmentation in the implementation of an object is a significant characteristic of an impedance mismatch. We label this kind of impedance mismatch an *instance mismatch* and it is addressed using an ORM transformation strategy. A programmer must reconcile fragmentation when developing an object-relational application.

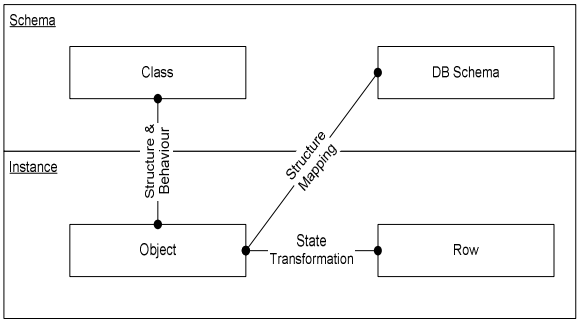

Figure 2. Fragmentation of the subdivisions of an object

The degree of state fragmentation that is characteristic of an instance mismatch is influenced by the ORM mapping strategy employed to produce the structure of an SQL schema. The design of that SQL schema is influenced by the ORM pattern strategy chosen to address a representation mismatch. Choices made within a level of our framework therefore have consequences in other levels.

An instance mismatch transformation strategy must address the consequences of fragmentation in behaviour and state. Such a strategy must deal with the processing model, encapsulation, and instance problems. The SQL standard does not provide support for the definition of behaviour within an SQL schema although relational database vendors have provided such facilities for some years. The valid state of object data may be enforced by rules defined within a class method or as a database constraint. Ambler ([5],p228) describes shadow information that is one strategy for the fragmentation of state, and scaffolding attributes that are one strategy for the fragmentation of structure.

The novel perspective provided by our framework produces new insights in areas such as how to exploit the strategic options available when translating a concept between paradigms and latent issues in solutions that cross over levels of abstraction. In providing an understanding of the issues with an ORM strategy based on levels of abstraction, our framework should provide standards bodies, tool vendors, designers and programmers with new insights into how to address problems of impedance mismatch both at the most appropriate level of abstraction and in the most appropriate way.

VI. A FRAMEWORK BASED APPROACH

One objective of our framework is to understand the issues and implications of a particular ORM strategy ("strategy"). Our framework does not assume that an object model drives the development of a database schema, nor does it prescribe where to start the analysis of a strategy. In the rest of this paper we show how to use our framework to understand the issues and implications of a strategy and what we can do about them.

Figure 3 is an outline of a four-stage process that provides context and guidance for the use of our framework. Our framework is concerned with the artefacts of object-relational design. The process uses our framework to identify issues with a strategy and to frame solutions to these issues. As such the process augments any software development cycle at the point where a choice of strategy must be made.

The process provides guidance for a change in the way we think about a strategy. Following the process shifts our thinking about a strategy from issues of implementation within the ORIM problem space into the new space provided by our framework. Our framework asks that we think about a strategy in terms of different levels of abstraction. This perspective facilitates new insights into a strategy, an understanding of cause and effect, and suggestions for improvements at the most appropriate level of abstraction.

Our process starts with the strategies in the ORIM problem space. Each strategy addresses one or more problems in the implementation of an object-relational application (Table I). The existence of a problem is the main driver for the use of a strategy. The literature provides some guidance on a choice of strategy based on costs and benefits. Future choices will also be informed by the outcomes from using our framework. The process then proceeds clockwise through the stages of comprehending a strategy, analysis of that strategy, understanding cause and effect in relation to issues with that strategy, and finally reflecting on the issues and suggesting changes to the strategy or the context in which it operates.

In the following sections (A through D) we describe each of the stages of our process. We show that using our process to understand available strategies facilitates a more informed choice of strategy. The objective of the first stage of our process is to comprehend a chosen strategy.

A. *Comprehend a Strategy*

The issue to be explored is how a strategy achieves its objective. In the first instance this comprehension will be based on the published literature and practical experience. We illustrate our approach using a case study that provides a context for strategy analysis. A case study helps clarify the semantics of a strategy, explain issues and highlight outcomes. Applying a strategy to a case study provides a worked example, demonstrates comprehension and cements understanding. A case study and worked example also provide material for illuminating issues in other stages of the process. Once we have an understanding of a strategy our process asks that we now move from the ORIM problem space and think in terms of our framework. In the next stage of the process we use our framework to analyse a strategy.

B. *Analyse a Strategy*

The objective is to provide new insights into a strategy. Issues to be explored include: whether a strategy is consistent in terms of our framework, whether a strategy correctly represent a data structure and its semantics, and whether the assumptions a strategy involves are safe assumptions to make. The resulting issues are then structured in terms of our framework.

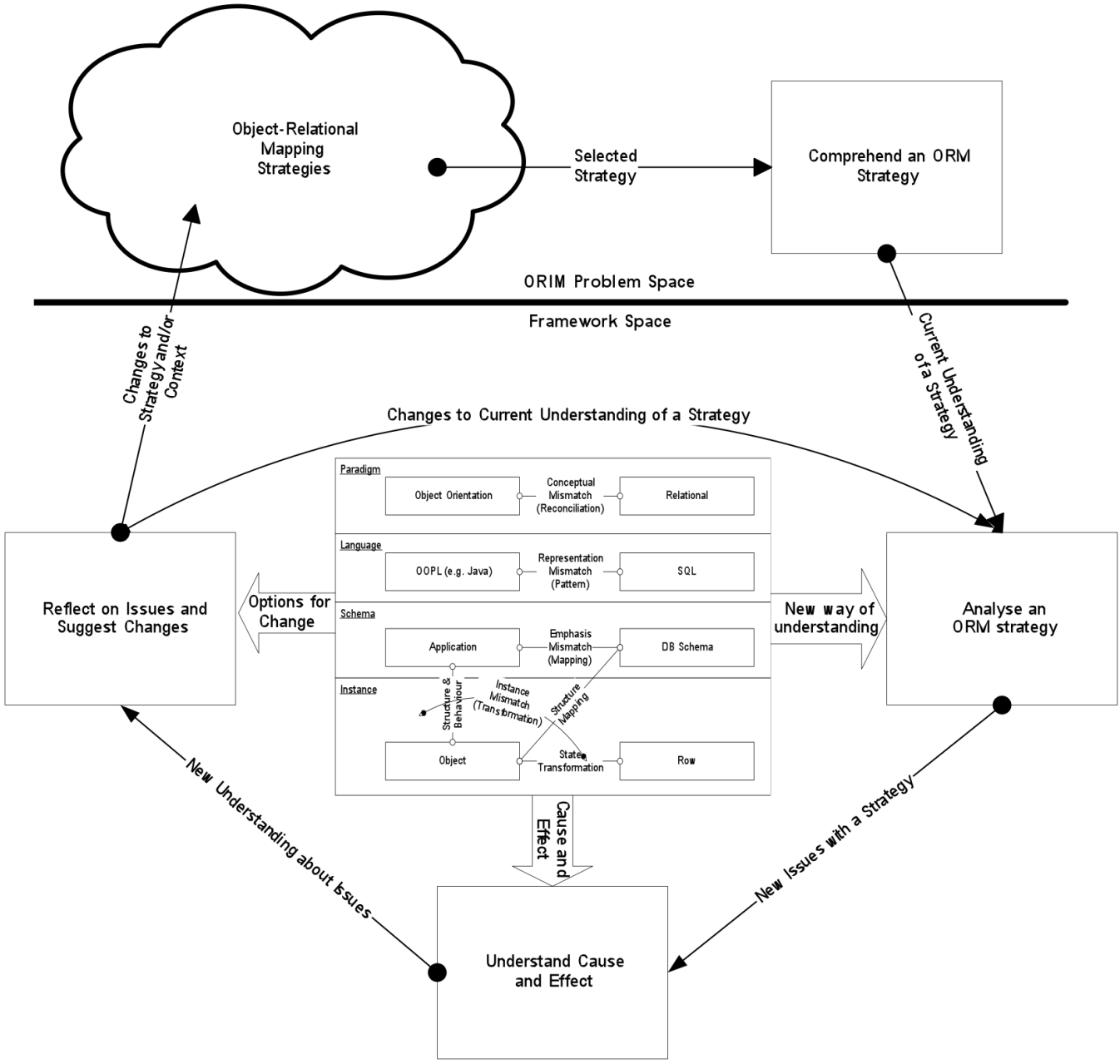

Figure 3. A Framework Based Approach

The issues must be phrased using terms at the appropriate level of abstraction. Each level provides a different focus for analysis and hence, a different set of terms (Table III). For any discourse between silos to be consistent and valid it is important that the corresponding set of terms are used. In the next stage of our process we identify the cause, effect and consequences of these issues.

TABLE III. GUIDELINES FOR THE TERMS USED WITHIN EACH LEVEL OF OUR FRAMEWORK

Level	Terminology
Conceptual	Terms relating to a particular world-view, irrespective of how it is actually described or implemented. Example terms include class, object, message, relation, tuple and union.
Language	Terms relating to language semantics, syntax and grammar, irrespective of a design. Example terms include UML class, Java Class, SQL table and column.
Schema	Terms relating to specific design choices including anything from a universe of discourse. Example terms include Order, Customer, Trade and Equity
Instance	Terms relating to data values. Example terms include instance, row, value and cast.

C. Understand Cause and Effect

Our framework is used to provide structure both to the analysis and the results. Issues to be explored include: whether an issue is related to the strategy or the context in which the strategy operates, whether the issue is local to a particular level, and if not what is the cause of an issue. An issue at the schema level may for example, be caused by an omission at the language level. This omission may be a limitation of a particular language or it may be caused by a conceptual difference. Such a conceptual difference would be beyond the scope of an object-relational application project to resolve. In order to correctly address a conceptual difference, the discourse would need to involve at least product vendors, standards bodies and possibly research bodies. Our framework provides the structure necessary to correctly identify and communicate both the cause and the consequences.

D. Reflect on Issues and Suggest Changes

Once we understand cause, effect and consequences we are in a position to suggest improvements to a strategy or to the context in which that strategy operates. In the final stage of our process we use the framework to identify options for change. Each level of our framework provides an opportunity to address an issue in a different way. The issues to be explored include: whether it is appropriate to make an improvement at a particular level, what change we need at that level in order to resolve an issue, and whether we change the strategy such that an issue is avoided. In order that others benefit and to avoid wasted effort, suggestions and improvements should be fed back into the wider discourse through changes to ORM tools, standards and the patterns used to describe strategy. Through the use of common abstractions and consistent terminology, our framework provides the structure necessary to communicate these improvements and to facilitate a coherent discourse for their implementation across cultures [23].

In the next section we provide an example of how the process and framework are used together to understand and improve a strategy.

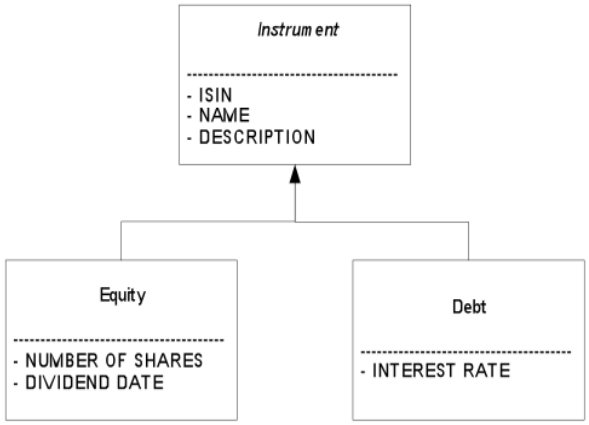

Figure 4. Financial Instrument Class Hierarchy

VII. USING OUR PROCESS – A WORKED EXAMPLE

Figure 4 presents a small class hierarchy for a financial instrument that provides a case study for our worked example.

There are two distinct and mutually exclusive kinds of instrument: Equity and Debt. Each is identified by an International Securities Identifying Number (ISIN) code. The ISIN code is defined under ISO 6166 and is unique across all financial instruments. In order to simplify the example, no associations or aggregates have been used.

We anticipate that such a hierarchy would form the basis for a Java application that would maintain data about financial instruments. The design of that Java application is beyond the scope of this paper but for now we will assume that Figure 4 provides a suitably accurate description of the class model.

Our requirement is to provide a means to store data about the objects of class Equity and class Debt in a relational database. We need to produce a data structure using SQL that corresponds to a Java data structure based on Figure 4. There are a number of strategies that take as their starting point a class hierarchy and produce an SQL-92 based representation. Three such strategies are [13] p9-17:

a) A single table per class hierarchy

b) A single table per concrete class

c) A single table per leaf class

Let us consider strategy (a) ("the strategy"). This strategy combines the definition of all classes in a hierarchy to form a single SQL table. A row of this table will store data about an instance of a class in the hierarchy. We are considering this strategy because Keller [12] recommends it as a strategy for a small application and

Ambler [24] recommends it for systems with a shallow class hierarchy. Ambler [5] suggests that during the development of an object-relational application refactoring is used to implement a change of strategy should it prove necessary. We will use SQL-92 for our example because no description of this strategy uses the additional facilities available in later versions of SQL.

In the following sections we show how in the context of our process, our framework is used to understand and improve the strategy. The outcomes of using our strategy could be used to compare strategies in order to choose the most suitable one. We do not show such a comparison in this paper but focus instead on improving a strategy.

A. Comprehend the Strategy

In the previous section we established our rationale for using the strategy. Here we show how the strategy achieves our objective to store object data.

1) Applying the Strategy to our Case Study

The process of applying the strategy is summarised as follows:

- Create a single table (Ambler suggests using the name of the root class as the table name).
- Create a column for each attribute.
- The data type of a column must correspond to the type of an attribute in so far as it must accept all possible values of that attribute.
- Each column representing a subclass attribute must accept NULL regardless of its definition in the class model.

Applying the strategy to Figure 4 produces the SQL-92 table definition presented in Figure 5. Note that a single row in this table will represent data about either an object of class Equity or an object of class Debt. The columns NUMBER_OF_SHARES, DIVIDEND_DATE and INTEREST_RATE must therefore accept NULL even though for their respective classes they are mandatory.

2) Assumptions

Descriptions of the strategy in Keller [13] and Ambler [24] make the following assumptions:

- It is *not* necessary to maintain the parent-child relationship between a class and a subclass in the database. This relationship is used to identify the attributes necessary for the definition of a table.
- An object can be fully described using a single row.
- The data types of a class attribute and a column are compatible.
- Only that column corresponding to an attribute of a class to which an instance belongs is set for a row. All other columns will be set to NULL.
- The mapping of a class attribute to a column is documented in some form or at least is somehow known by those who must use it.
- If the data type of a class member attribute is changed, regardless of the topological position of the class in the hierarchy, that change applies to all rows of the table.

3) Costs and Benefits

The main benefits of this strategy are [24]:

- Data about all objects is accessible from a single table;
- There is only one table for a programmer to consider;
- The mapping from a class hierarchy to a single table is a "simple approach";
- It is easy to add a new class should requirements change.

```
create table INSTRUMENT(
   ISIN CHARACTER(12) PRIMARY KEY,
   NAME CHARACTER(20),
   DESCRIPTION CHARACTER(40),
   NUMBER_OF_SHARES INTEGER NULL,
   DIVIDEND_DATE DATE NULL,
   INTEREST_RATE FLOAT NULL)
```

Figure 5. The SQL-92 table derived from the Instrument class hierarchy

Ambler and Keller describe a number of issues with this strategy. One such issue relates to classification. In order to maintain the class member semantics of a collection of data values in the context of table INSTRUMENT, there must be some means to differentiate between data for an object of class Equity and data for an object of class Debt. There are at least three options for achieving this:

a) Infer the class of a row from the existence of values for its attributes [13], p13. For example only the row for an object of class Equity will have a value for NUMBER_OF_SHARES. For a Debt object this column would have a NULL value.

b) Augment the table definition with a new column the value of which indicates the class of data to which a row belongs [24]. For a row representing data about an object of class Equity for example, this column would have the value "EQUITY".

c) Use a discriminator value from the universe of discourse in order to differentiate the class of data stored in a row [13], p13. Similar to option "a" but here we look at the actual value not its presence, and option "b" but use the values stored in an existing column rather than creating a new one. All Debt ISIN codes could include the character "D" at a certain position. This character indicates that the ISIN code is non-atomic and identifies data about an object of class Debt.

Let us consider option (a) because it does not require the maintenance of additional data. A user of the table INSTRUMENT must know how to infer class membership.

Other issues documented in [13] and [24] include potential wasted space in the database through the use of NULL, the consequences of certain changes to the class

hierarchy, locking issues because all data is the same space, and indexing issues because secondary indexes are required.

With as full grasp of the current knowledge about a strategy as space allows, and a worked example based on a case study to cement this understanding, we are now in a position to start our analysis. This analysis will identify the cause of these implementation issues and highlight new issues.

B. Analysis of the Strategy

The observations and insights we have gained during our analysis of the strategy are described in the following subsections. They are not listed in any particular order but they are categorised using the levels of our framework.

1) Conceptual Insights

We need to represent a class hierarchy in a SQL schema in order to provide for requirements and design traceability. It is essential to understand the semantics of a particular class hierarchy before applying the strategy. The strategy does not make clear which definition of a class is being used.

In mapping a class to a table, the strategy mixes levels of abstraction (the shaded boxes in Figure 6). The term "table" is a language level construct within the relational silo. The term "class" is a conceptual level construct within the object silo. This strategy should either map a class to a relation or map a class in a particular object-oriented programming language, e.g., Java to a table. This is an important distinction because the semantics of a Java class are not the same as those of a C++ class at the language level. A C++ class for example supports multiple inheritance.

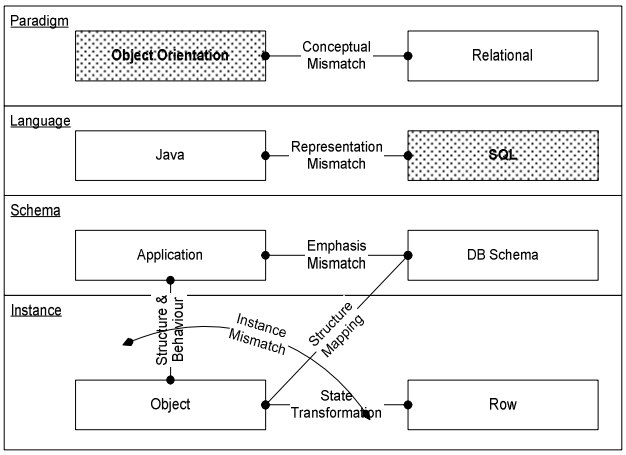

Figure 6. Mixing levels of abstraction

A relation represents a kind of fact. In combining the definitions of all subclasses into a single relation this strategy overloads the semantics of a relation. A relation must now represent more than one kind of fact although each tuple represents a single fact.

2) Language Insights

The strategy employs a class hierarchy as the basis for the definition of a table, but the actual hierarchy is not represented in the database. We therefore lose requirement and design traceability.

SQL-92 provides no explicit support for a hierarchy in the definition of a table. Support for a hierarchy can be designed into a table but the strategy does not require, nor the SQL-92 based representation (Figure 5) preserve, the parent-child relationship between a class and a subclass.

To ensure data integrity and to enforce the semantics of a disjoint subclass, there must be some way of identifying to which class the data in a row belongs.

A column in a table represents an attribute of a class. The assumption is that they are of equivalent data types. SQL-92 has a predefined set of data types. The type of a class attribute may be another class although that is not shown in this example. The definition of that class is a schema level decision so there is no guarantee of type compatibility at the language level and casting must be used. The strategy does not describe how to address differences in type or scale.

3) Schema Insights

The classes Equity and Debt are disjoint. Class Instrument is abstract. These are design features built into our class model (Figure 4). The SQL table INSTRUMENT is formed from the union of the attributes of the class Instrument and the subclasses Equity and Debt. These attributes are represented as columns of the table INSTRUMENT. Data about each object is stored in a row of the table INSTRUMENT. As the primary key, ISIN provides the semantics of a disjoint subclass because it is unique across all financial instruments.

Instrument is an abstract class. Although no object of this class *should* exist, for reasons of data integrity it is important to prevent the insertion of a row of this class in the table INSTRUMENT.

The data type of each column of the table INSTRUMENT has a trivial correspondence to the type of the corresponding attribute of each class. This correspondence is not always so trivial. Some attributes may be derived or use names which are not the same as or similar to the column name. A user of the schema must understand that a row of the table INSTRUMENT represents data about one of two kinds of object. They must also know how to differentiate those kinds of data.

4) Instance Insights

The class of data stored in a row may be determined in a number of ways. The choice must be made clear to those who use the table. In this example we have chosen to infer the class from one or more column values.

The semantics of the Instrument class hierarchy are not represented in the table INSTRUMENT and so it is not straightforward to query over a subclass of Instrument and all its subclasses. In our simple example classes Equity and Debt do not have a subclass. If they did a programmer must understand the conditions for returning only those

rows belonging to each object of each subclass in which they have an interest.

TABLE IV. INSIGHTS INTO THE STRATEGY FROM USING OUR FRAMEWORK

Level	Insights into the Strategy
Conceptual	Mix levels of abstraction. A relation has no explicit semantics of hierarchy. Overload the semantics of a relation.
Language	Omit the subclass relationship. Issues of type or scale between an attribute and a column.
Schema	Enforce the semantics of a subclass. Enforce the semantics of an abstract class. Make explicit the correspondence between attributes and columns. Differentiate the class of data held in a row.
Instance	Identifying the class to which the data in a row belongs. Query a sub-hierarchy.

5) Summary

We have used our framework to question the strategy at a number of levels of abstraction. Table IV summarises the insights that thinking about the strategy in terms of our framework provides. Each level of our framework has focused attention on a different aspect of the nature of the strategy. Our framework has helped us to see new issues and relationships. In the next section we use our framework to explore these relationships and their consequences.

C. Understand Cause and Effect

Here we provide two examples of cause and effect relationships based on the conceptual issues of overloaded semantics and support for the semantics of hierarchy.

1) Overloaded Semantics

In Table V we use the levels of our framework to show the consequences of overloading the semantics of a relation.

The strategy is described in [13] p9-17 using terms that we generally recognise within the language level of our framework, for example a class corresponds to a table and an attribute corresponds to a column. This strategy does not address the root cause of this problem only the symptoms. Our framework shows that we must look to the conceptual level for the cause of the overloaded semantics problem.

The results of applying our framework (Table V) clearly show that the conceptual problem of representing more than one kind of fact using a relation has consequences within the levels below. The strategy confronts this conceptual problem at the language level by requiring a way to differentiate the data stored in a row. The choice of mechanism for differentiation will impact the definition of the table INSTRUMENT at the schema level. Ultimately at the instance level, a programmer working with data in a row of table INSTRUMENT must understand how to differentiate between data about an Equity object and data about a Debt object.

Using our framework we relate the consequences of this conceptual problem back to the implementation problem of wasted space described by Amber and Keller (see Section VII.A). The overloading of a relation necessitates NULL valued columns. WHERE clause complexity is another consequence of overloading the semantics of a relation.

TABLE V. OVERLOADING THE SEMANTICS OF A RELATION

Level	Consequences
Conceptual	In our example a relation must represent two kinds of fact. There must be some way to differentiate the class of data held in a relation.
Language	An SQL table is a representation of a relation. The SQL language requires that all rows in a table share the same definition provided by that table. The strategy necessitates we compromise by (i) providing some way to differentiate a row, and (ii) accepting that a column corresponding to a subclass attribute must accept NULL. Preservation of semantics requires that the SQL language support a form of constraint. There must be a mechanism to document the correspondence between a column and a class attribute.
Schema	The definition of table INSTRUMENT must provide some way to differentiate a row. A row may represent data about an object of class Equity and an object of class Debt. We chose to infer class membership from column values. The columns NUMBER_OF_SHARES, DIVIDEND_DATE and INTEREST_RATE must accept NULL even though for their respective classes they are mandatory. A database constraint must ensure attributes are populated correctly based on class.
Instance	Our chosen option for identifying class membership does not require additional columns but class membership is not explicit. A user of the table INSTRUMENT must understand how to differentiate the class of a row based on the value or one or more columns. Differentiating a row based on multiple columns adds complexity to a WHERE clause. Using another method for identifying class membership requires more data be maintained but would make class membership more explicit. A program must ensure that the correct columns are populated for each subclass. A DBA must enforce this using a constraint. The use of NULL values can result in wasted space in a database but this depends on vendor implementation.

2) Omitting the Semantics of Hierarchy

In Table VI we use the levels of our framework to show the consequences of omitting the semantics of hierarchy.

We must look to the conceptual level of our framework for the root cause of the hierarchy problem. The semantics of hierarchy are not present in a relation. The strategy does not attempt to address this problem at the language level. As a result at the schema and instance levels it is necessary to encode the semantics of a hierarchy outside the table INSTRUMENT.

A consequence of omitting the semantics of hierarchy is that these semantics are encoded in database constraints and in each query that needs to make use of them. Should the hierarchy change, all places where these semantics are encoded must also be identified and changed. We must encode the semantics of hierarchy because they are not

present in the table INSTRUMENT. They are not present because an SQL table does not have explicit support for a hierarchy and the strategy does not address this. An SQL table has no support for hierarchy because it is based on the concept of a relation which itself supports no notion hierarchy. If we had adopted the approach of [3] and focused solely on aspects covered by the schema an instance levels of our framework, we would not have identified the real cause of this problem.

We have identified new issues, traced their cause, and shown that they have consequences for those developing an object-relational application. Our framework can also be used to understand an existing issue and to provide one possible chain of cause and effect. Table VI shows that whilst providing a means to query the Instrument hierarchy, this strategy introduces problems if one wishes to query any hierarchy below that (an issue not present in our case study).

TABLE VI. OMITTING THE SEMANTICS OF HIERARCHY

Level	Consequences
Conceptual	The semantics of a class hierarchy are well defined, but the actual semantics in use depend on the context provided by a class model and the intention of the designer. A relation has no explicit semantics of hierarchy.
Language	An SQL-92 table also has no explicit semantics of hierarchy. The strategy does not provide a means to address this.
Schema	The semantics of the Instrument class hierarchy are not present in the table INSTRUMENT. In order to preserve data integrity, a DBA must encode these semantics in one or more database constraints.
Instance	In terms of the class hierarchy, all we can say about a single row is that it belongs to a given class and to the hierarchy rooted at class Instrument. Information regarding the topological position of that class in the hierarchy is not present in either the data or the definition of the table INSTRUMENT. A position may be inferred [25] but this should not be necessary and is prone to ambiguity. As a consequence, to correctly form a polymorphic query over a sub-tree, a programmer must encode the semantics of the Instrument hierarchy in a query. The deeper the hierarchy one represents using a single table, the more complex the WHERE clause becomes. This is particularly true if one wishes to query data for objects belonging to a leaf class.

Our framework can also be used to clarify received wisdom. Contrary to Keller's suggestion in [13], p13, it is not sufficient to only identify to which class a row of data belongs. The query must also include the semantics of that hierarchy. Table VI uses our framework to show why a query must include these semantics.

The root cause of an issue is not always at the conceptual level of our framework. We have assumed that there is a direct correspondence between a class hierarchy and a table. A schema provides the context necessary for normalisation. Normalisation is a process within the relational silo that breaks down correspondence at the schema level. This issue must be resolved within the design of a schema.

3) Summary

We have shown that our framework provides a way to understand both the cause of an issue with a strategy and the consequences of that issue. The root cause of an issue may be at any one of the levels of our framework and its effect may materialise in different ways. In the next section we use our framework to reflect on this new understanding and suggest opportunities for improvement.

D. Reflect on Issues and Suggest Changes

We have identified two issues with the strategy: overloading the semantics of a relation, and omitting the semantics of hierarchy. We can improve the strategy in two ways: either indirectly by addressing the symptoms of an issue or directly by addressing the context.

The context of any given level of our framework is those levels above it, so for the schema level the language and conceptual levels provide the context. The cause of an issue may be at any level of our framework. Our framework also provides a means to understand at which level symptoms emerge and for thinking about the most appropriate approach to address them.

Understanding cause and effect is not the only requirement for change. The ability to effect change depends on the power and influence of those involved. Ideally the root cause of an issue should be addressed, but this is not always an option for those developing an object-relational application. Their influence will typically concern the schema and instance levels although the use of dynamic languages such as Ruby and Groovy [26] may change this. If an issue is best resolved at the conceptual or language levels they will still have to adopt an indirect approach and therefore only address the symptoms of an issue. Those involved with the definition of a standard or the design of a programming language will have influence to affect change at the language level. Research bodies and the community in general are best placed to deal with a conceptual issue. They have the power and influence to adopt a direct approach.

1) Indirectly

An indirect approach takes context as given and will not address the root cause of an issue. A solution at the schema level must work within the constraints of the languages used and as a result also accept any conceptual problems. The root cause of both our issues is at the conceptual level. A direct approach in this case will therefore involve avoidance or mitigation at best. Table VII summarises some of the indirect options available for addressing aspects of each issue.

Ultimately it may be more appropriate to use a different strategy. In order to address the first issue we could use a strategy that involves creating a separate relation for each concrete class [24]. This would remove some of the WHERE clause complexity in terms of class identification and joins, wasted space and the need to

maintain additional data but would go against the spirit of the strategy: to represent all data in a single table.

TABLE VII. INDIRECT OPTIONS

Level	Suggestions
Schema	Use a different strategy. One that produces a separate relation for each concrete class. Create a database view for each class. Add a column PARENT_CLASS that indicates the parent class.
Instance	Using a different strategy avoids maintenance of additional data. Use a database view to realise data for a subclass. Infer class membership from attribute values. Set the value of PARENT_CLASS to be the classifier value for the parent class.

One solution that does not involve a change of strategy is to retain the single base table INSTRUMENT but represent each subclass or subclass hierarchy as a database view. Using a database view *hides* WHERE clause complexity for a schema user and the semantics need only be defined in one place. This solution does not address the need to maintain additional data or the problem of wasted space in the base table (although this is arguably a database vendor implementation issue). The use of a database view would increase the space required but only marginally if a materialised view is not stored. We can avoid the maintenance of additional data if we continue to infer class membership from the existence of data values or use an existing discriminator value from the universe of discourse.

Neither approach addresses the omission of the semantics of hierarchy. Adding a column PARENT_CLASS does not solve the problem because it confuses intent and extent. The semantics of a hierarchy are mixed with the data representing an object. This messy implementation fudge is not a viable solution because it is still necessary to know how the hierarchy is structured and there are problems with an abstract class or any class where no rows (yet) exist.

2) Directly

Here we use the levels of our framework to suggest changes to the context in which the strategy operates. In Table VIII and Table IX are options for addressing both issues at each level of our framework. We do not propose a complete solution. Our objective is two fold. First to show that there are options at the conceptual and language levels, and second to highlight that these provide different options at other levels of our framework.

In the case of both issues, the root cause of the problem is at the conceptual level of our framework. This is therefore the most appropriate level at which to make improvements, but changes at this level are the most fundamental. A change at the conceptual level will have far reaching consequences, will require input from researchers and standards bodies, and consequently will take time to implement. Such a change is out of scope for any object-relational application development project.

We note that work to address these issues by changing context has already started. The majority of the solution described in Table IX is possible using the object-relational features introduced in SQL:1999 ("OR-SQL"). Only the ability to insert data into the table INSTRUMENT and have a row created in an appropriate sub-table is not supported. Although counterintuitive, this facility may be important for a programmer because it maintains the single table nature of the solution provided by the original strategy.

TABLE VIII. DIRECT OPTIONS FOR ADDRESSING OVERLOADED SEMANTICS

Level	Suggestions
Conceptual	Recognise that a relation may represent more than one kind of fact.
Language	Provide a classifier mechanism in the definition of a table. Extend the SQL language or its implementation to support optional columns based on this classifier.
Schema	Do not represent Equity and Debt as subclasses. Use a single class Instrument. This is not in the spirit of the object model and may cause issues in the object silo. Represent each class using a separate table. Again, this is not in the spirit of the strategy.
Instance	Provide access to the classifier mechanism above within a query. Insert only the data values required based on the classifier. Omit a column if it is not relevant to a particular kind of row.

In Section VII.A.3 we listed some of the benefits of the strategy. These benefits come at a cost. Storing data about all objects in a single table may be a "simple approach" [24] but it has costs in terms of work on database constraints and queries. Whilst it may be easy to add a new class, such a change has consequences including the maintenance of database constraints and queries. Our framework has drawn attention to these problems and provided a way to think about improving the situation. The information emergent from the use of our framework and process is therefore of benefit to those who must choose and implement this strategy.

TABLE IX. DIRECT OPTIONS FOR ADDRESSING THE OMISSION OF HIERARCHY

Level	Suggestions
Conceptual	Recognise the possibility of a hierarchy of relations. Support the concept of an abstract relation.
Language	Support a hierarchy of tables and permit a single query over the hierarchy of tables. That query does not need to include the names of all sub-tables.
Schema	Create a separate table for classes Instrument, Debt and Equity but each table is part of a hierarchy of tables. Each table may be queried individually or as part of a hierarchy.
Instance	Create a row in the corresponding base table or by inserting into table Instrument. Query the entire hierarchy or part thereof using a single statement.

3) Summary

Our analysis has demonstrated that this ORM strategy does not address two conceptual problems because it is a

solution at the language level of our framework. The strategy does not attempt to mask these problems and this results in work for those who use it to implement an object-relational application.

There are indirect options open to those developing an object-relational application. Whilst these do not address the fundamental problem they will improve the situation in the short term leaving time for conceptual issues to be addressed through a direct approach.

There are a number of strategies for any impedance mismatch problem. It may be that using another strategy is more appropriate for those developing an object-relational application. We anticipate that effecting change at the conceptual and language levels of our framework will be more difficult than at the schema or instance levels. Changing the definition or implementation of SQL for example is not feasible for those developing an object-relational application. Our framework provides a basis for making the decision to change by asking that we think about cause, effect and consequences. That information helps when selecting amongst alternatives. At this point we have come full circle in our process (Figure 3).

VIII. Conclusion and Future Work

Our conclusions concern the framework used to understand a strategy and the process by which we used the framework to suggest improvements to a strategy.

1) *The Framework*

We have demonstrated that understanding a strategy at different levels of abstraction does identify the root cause of an issue. Our framework is not concerned with the issues of implementation that have driven work by Ambler [24] and Fussell [11]. We have also demonstrated that in order to address an ORIM problem at the most appropriate level of abstraction we must understand the real issues that underpin that problem.

In our framework we have a new way to understand an ORM strategy. If we think about a strategy at a number of levels of abstraction we find new insights into a strategy. These insights provide an opportunity to improve a strategy and the context in which a strategy operates. If the outcomes appear obvious it is because of the new perspective provided by our framework. A perspective that takes context as given, is driven by a single problem, or which views a solution as an exercise in software architecture ([16], [11]) will not produce the same results.

Ambler [16] suggests software architecture as a means to shield a programmer from the details of a strategy. In terms of our framework this is predominantly a schema level activity within the object silo. Fussell [11] suggests a separation based on client and server. This separation corresponds loosely to the object and relational silos of our framework. Fussell's emphasis is on decoupling but impedance mismatch problems occur when we try to combine object and relational artefacts. Neither perspective provides the same scope or a means to facilitate an analysis of cause and effect and an understanding of consequences that we have achieved from the use of our framework. Taking a step back from the detail of implementation, our framework allows us to address the cause of a problem, not its symptoms, at the most appropriate level of abstraction.

The information elucidated through the use of our framework will be of use to standards bodies, tools vendors and those who define a strategy. Thinking about the consequences of a strategy provides information necessary to choose between alternatives. Those working on an object-relational application can now make a more informed choice of strategy. Those working on database and programming language standards see the impact of past choices and the need for change. Researchers in object and relational concepts see the consequences of their work and that there is still work to be done to cross the chasm [27].

The framework helps bridge the cultural impedance mismatch [23]. Through the use of common levels of abstraction our framework facilitates a discourse between proponents of object and relational perspectives. A specific set of terms must be employed at each level of the framework although further work is required to develop a formal ontology of terms based on Table III. We are now in a position to address problems of an ORIM in a structured and consistent way, not just across levels of abstraction but also between silos. We can now think in an integrated way, for example how decisions made in the design of Java correspond to structures in SQL or vice versa. We also have a way to understand the impact of these changes for those designing both an object and a relational schema and programming an object-relational application.

Another opportunity for our framework is to understand the impact and potential of changes introduced in OR-SQL on the current ORM strategies. In terms of our framework, OR-SQL appears to characterise a language level change in the relational silo. Further work is required to understand the opportunities these changes present for new or enhanced ORM strategies with languages such as Java, LINQ [15] and Ruby [26].

A generalised form of our framework could help to understand issues at the junction of any two paradigms in computing or other disciplines.

2) *The Process*

We have demonstrated that our process provides the necessary guidance to improve a strategy. We have identified options for change that are linked to a conceptual problem not a symptom of an implementation. We have also demonstrated that our process supports a shift in thinking away from implementation issues because we start by understanding a strategy and issues of implementation, but finish by suggesting solutions at a number of levels of abstraction.

References

[1] Ireland, C., Bowers, D., Newton, M., Waugh, K.: A Classification of Object-Relational Impedance Mismatch. In: Chen, Q., Cuzzocrea, A., Hara, T., Hunt, E., Popescu, M. (eds.): The First International Conference on Advances in Databases, Knowledge and Data Applications, Vol. 1. IEEE Computer Society, Cancun, Mexico (2009) p36-43

[2] Neward, T.: The Vietnam of Computer Science (http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx) (6th February 2007)

[3] Stathopoulou, E., Vassiliadis, P.: Design Patterns for Relational Databases. Vol. 2009. ODMG (2009)

[4] Meijer, E.: There is No Impedance Mismatch (Language Integrated Query in Visual Basic 9). OOPSLA. ACM, Portland, Oregon (2006)

[5] Ambler, S.W.: Agile Database Techniques - Effective Strategies for the Agile Software Developer. Wiley (2003)

[6] Keller, A.M., Jensen, R., Agarwal, S.: Persistence Software: Bridging Object-Oriented Programming and Relational Databases. In: Buneman, P., Jajodia, S. (eds.): ACM SIGMOD international conference on management of data, Vol. 22. ACM Press, Washington, D.C (1993) 523-528

[7] Marguerie, F.: Choosing an object-relational mapping tool (http://weblogs.asp.net/fmarguerie/archive/2005/02/21/377443.aspx) (14th November, 2007)

[8] Holder, S., Buchan, J., MacDonell, S.G.: Towards a Metrics Suite for Object-Relational Mappings. COMMUNICATIONS IN COMPUTER AND INFORMATION SCIENCE **8** (2008) 43-54

[9] Hibernate: (www.hibernate.org)

[10] Biswas, R., Ort, E.: The Java Persistence API - A Simpler Programming Model for Entity Persistence (http://java.sun.com/developer/technicalArticles/J2EE/jpa/index.html) (25th September 2007)

[11] Fussell, M.L.: Foundations of Object Relational Mapping (http://www.chimu.com/publications/objectRelational/) (25th September 2007)

[12] Hohenstein, U.: Bridging the Gap between C++ and Relational Databases. In: Cointe, P. (ed.): European Conference on Object-Oriented Programming, Vol. Lecture Noted on Computer Science 1098. Springer-Verlag, Berlin (1996) 398-420

[13] Keller, W.: Mapping Objects to Tables: A Pattern Language. In: Bushman, F., Riehle, D. (eds.): European Conference on Pattern Languages of Programming Conference (EuroPLoP), Irsee, Germany (1997)

[14] Lammel, R., Meijer, E.: Mappings Make Data Processing Go 'Round: An Inter-paradigmatic Mapping Tutorial. Lecture Notes in Computer Science **4143** (2006) 169-218

[15] Schwartz, J., Desmond, M.: Looking to LINQ (http://reddevnews.com/features/print.aspx?editorialsid=707) (23rd October 2007)

[16] Ambler, S.: The Design of a Robust Persistence Layer for Relational Databases (http://www.ambysoft.com/downloads/persistenceLayer.pdf) (10th May 2007)

[17] Griethuysen, J.J.v. (ed.): Concepts and Terminology for the Conceptual Schema and the Information Base. ISO, New York (1982)

[18] Coad, P., Yourdon, E.: Object Oriented Analysis. Yourdon Press (1990)

[19] Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM **13** (1970) 377-387

[20] Kalman, D.: Moving forward with relational: looking for objects in the relational model, Chris Date finds they were there all the time. DBMS, Vol. 7 (1994) 62(66)

[21] Meijer, E., Schulte, W.: Unifying Tables, Objects, and Documents (http://research.microsoft.com/~emeijer/Papers/XS.pdf) (21st August 2007)

[22] Sutherland, J., Pope, M., Rugg, K.: The Hybrid Object-Relational Architecture (HORA): an integration of object-oriented and relational technology. ACM/SIGAPP symposium on Applied computing: states of the art and practice. ACM Press, Indianapolis, Indiana, United States (1993)

[23] Ambler, S.: The Cultural Impedance Mismatch Between Data Professionals and Application Developers (http://www.agiledata.org/essays/culturalImpedanceMismatch.html) (10th May 2007)

[24] Ambler, S.: Mapping Objects to Relational Databases: O/R Mapping In Detail (http://www.agiledata.org/essays/mappingObjects.html) (12th April 2007)

[25] An, Y., Borgida, A., Mylopoulos, J.: Discovering the Semantics of Relational Tables Through Mappings. LNCS 4244 - Journal on Data Semantics VII (2006) 1-32

[26] Richardson, C.: ORM in Dynamic Languages. Communications of the ACM **52** (2009) 48-55

[27] Brown, K., Whitenack, B.G.: Crossing Chasms: A Pattern Language for Object-RDBMS Integration "The Static Patterns" (http://www.ksc.com/articles/staticpatterns.htm) (30 December 2008)