2. Literature Review

During the past decade, the requirements for data storage have changed from short and simple flat records to complex nested data, compound data, and multimedia data. Current relational database systems are hard-pressed to accommodate these various data while maintaining the productivity and performance requirements. Also, the widespread adoption of object-oriented technologies has fuelled the need for a persistent object storage mechanism.

As opposed to the simple data model used by traditional business applications using a relational database management system (DBMS), object-oriented applications make extensive use of many new object-oriented features such as a user-extensible type system, encapsulation, inheritance, dynamic binding of methods, complex and composite objects (not first-normal-form objects), and object identity. The limitations of the data models supported by the relational DBMS therefore need to be relaxed in order to enable the building of more complex (object-oriented) business and non-business applications. As a result, there has been much activity in designing and implementing systems to handle object persistence.

The object-oriented database systems available today are either persistent storage managers for object-oriented programming languages (OOPLs) or object-oriented databases with full ANSI SQL support. They are commonly referred to as first and second generation object-oriented databases respectively. A persistent storage manager for an OOPL is a file system that automatically stores on disk each object created in an object-oriented program. After the program terminates, it automatically fetches any stored object on demand by another program.

2.1 The Relational Model

Current relational databases in use are mostly based on the relational model, which consists of three parts:

- Data structures called tables or relations.
- Integrity rules (entity and referential integrity) on allowable values and combinations of values in tables.
• Data manipulation operators which can be either the relational algebra or the relational calculus plus an assignment operator. Typical operations are Select, Project, Product, Union, Intersection, Difference, Join and Divide.

A table is simply a two-dimensional array which contains data. Each row of the table corresponds to one item or record, and the columns are fields containing the data for the record. For example, a customer record contains the customer ID, name, address, phone number, etc. as its fields. An application will generally use a series of tables with multiple rows and fields to store its data.

In order to uniquely identify an item it must be assigned a value in one or more of its fields that is guaranteed to be unique within its table. This is the primary key of the table. For example, in a customer table, the 'customer ID' can be the primary key. If one item contains the primary key of another item then it is called a foreign key. Foreign keys allow relationships between items to be expressed. For example, an invoice table can contain a 'customer number' field which refers to the corresponding customer in the customer table for which the invoice was issued.

2.1.1 Limitations of Relational Databases

Relational DBMSs (RDBMSs) have a firm theoretical base and satisfy many applications. They are good at managing large amounts of data and in retrieving them. RDBMSs, however, lack important features needed for advanced applications, such as abstract data types, complex integrity constraints, and versioning. Programming languages, such as Pascal and C++, provide abstract data types, structured control constructs, and the ability to write complex algorithms, but lack data persistence across executions and concurrent access to data.

The relational model cannot capture and control the semantics of complex applications. For example, although the relational model enforces referential integrity, it has no mechanism for distinguishing between the different kinds of relationship which may exist between entities. Many different kinds of relationship exist. These include: one-to-one, one-to-many and many-to-many.

Relational DBMSs implement a limited number of data types, typically integer, real,
character, money and date. However, many applications require more complex object types. In turn, these types require new operations to be defined. Older RDBMSs tend to allow only insert, update, delete and retrieve operations. Some newer RDBMSs do allow for user-defined data types and operations, as well as storing of objects. These are also known as object-relational DBMSs.

2.2 The Object Model

The object model is based on objects, which is an analogy of how real world objects behave. An object is an entity that has state, behaviour, and identity. The structure and behaviour of similar objects are defined by their common class. A class can be defined as a specification for a set of objects with common structure and behaviour. Therefore an object is simply an instance of a class. Objects have the following three basic properties:

- **Encapsulation** – this shields the internal workings of the object from the user of the object. As such the user will only need to know how to interface with the object and changes can be made easily to the object without affecting its interface.

- **Inheritance** – this allows classes to be built using existing code and data in other classes. This allows common features of a set of classes to be expressed in a base class and inherited as necessary. For example, Car, Boat, and Plane classes can all inherit from a base Vehicle class.

- **Polymorphism** – this allows classes to respond differently to the same message. For example, the same Move() message can be passed to Vehicle objects Car, Boat, and Plane and each will respond appropriately. Therefore new vehicle types can be added later without having to change the message sent to it.

An object can also have relationships with other objects. In C++ relationships are expressed using pointers, which are memory addresses of the objects, and are used as the object’s unique identifier. An object which needs to reference a different object will do so by using a pointer to the other object. Pointers are very similar to the foreign keys used in relational databases, as described in the previous section.

2.2.1 Object-oriented Databases
Object-oriented DBMSs (OODBMSs) are intended to address the shortcomings with first and second generation DBMSs. They provide an object-oriented storage mechanism combined with a direct interface to an object-oriented programming language (OOPL). This is essentially an OOPL with support for persistent objects. In addition to all the normal database functions, OODBMSs also support complex objects, encapsulation, types and classes, inheritance, late binding and extensibility. These are the features proposed in the Object Oriented Database System Manifesto [2], [7], which can be summarized as:

- Complex objects must be supported.
- Object identity must be supported.
- Encapsulation must be supported.
- Types or classes must be supported.
- Types or classes must be able to inherit from their ancestors.
- Method dispatches should be bound dynamically.
- Any computable function must be expressible in the language.
- There should be no distinction between system-defined and user-defined types.
- Data persistence must be provided.
- It must be possible to manage very large databases.
- The DBMS must support concurrent users.
- The DBMS must be capable of recovery from hardware and software failures.
- It must be possible to state database queries in a high-level, concise form.

2.2.2 Objects and Relational Databases

In the real world, it may not be possible to have an ideal situation where objects can be stored into a readily available OODBMS. OODBMSs are rather young, compared to three decades of RDBMSs, which have been used as stable tested platforms for managing data for conventional transaction-oriented applications. As such, applications have to interoperate with older legacy databases, as well as newer object-oriented databases. Thus, an object-oriented application may have to store objects in a non-object-oriented database, which brings about the problem of mapping objects to RDBMSs.

Due to the shortcomings of RDBMSs mentioned previously, storing objects into it is
not a straightforward task. The object paradigm is based on building applications out of objects that have both data and behaviour, whereas the relational paradigm is based on storing data. This fundamental difference results in a non-ideal combination of the two paradigms; the so-called "impedance mismatch". Tradeoffs have to be made, depending on the type of mapping used.

For an object-oriented system, the important things that need to be stored are classes, its attributes, and its relationships to other classes. Classes can be mapped to tables, and its attributes mapped to columns in the RDBMS. This too may not be a perfect one-to-one mapping, depending on the classes to be mapped.

2.3 Mapping Inheritance

According to Ambler [5], Keller [11], and Objectmatter [10], there are three fundamental solutions for mapping inheritance in a class hierarchy into a RDBMS:

1. **Use one table for an entire class hierarchy** (also known as filtered mapping [10])

   This involves flattening out the class hierarchy into a single table; i.e. all the attributes of the classes in the hierarchy and its inherited superclasses are stored as columns in the relational table. In addition, a filter column is created in the table. The value of the filter column is used to distinguish between subclasses. Abstract classes are not mapped to this table. This approach provides adequate performance, but violates table-normalization rules. More specifically, it could lead to a substantial number of NULL columns in the table, wasting space. Consequently, this method is most useful if most of the attributes are inherited from the abstract parent classes.

2. **Use one table per concrete class** (also known as horizontal mapping [10])

   All class attributes, including inherited attributes are stored in a single table. Only classes which can be instantiated (non-abstract) are stored. Each of these mapped tables contains columns for all attributes in its concrete class, plus all attributes inherited from all its abstract parent classes. In other words, abstract classes are not mapped to their own table. This approach provides very fast performance, and is simple to design. However, if an attribute of an abstract parent class is changed, then potentially many tables must be modified. Consequently, this method is most useful if the inheritance tree is more method
driven than attribute driven. To be more specific, if a substantial number of classes inherit a large number of attributes from an abstract parent class, then the vertical or filtered methods might be better choices.

3. **Use one table per class** (also known as vertical mapping [10])

Each class is stored in one table, including abstract classes. This method conforms very well with object-oriented concepts as each table can be considered a class. This can be accomplished by means of a foreign key column that references the primary key of the

![Class hierarchy to be mapped](image)

**Figure 2.1** – Class hierarchy to be mapped

![Three different class inheritance mappings](image)

**Figure 2.2** – Three different class inheritance mappings
parent table. In order to instantiate a concrete class using this method, a join query of the
concrete class table and all its abstract parent class tables must be performed.

The above mappings are illustrated in Figure 2.1 and Figure 2.2. Figure 2.1 shows
the class hierarchy, and Figure 2.2 shows the mapped tables.

Brown and Whitenack [8] also suggest similar mappings, namely the horizontal and
vertical mappings. They further mention that horizontal mappings should be used when the
speed of queries is more important whereas vertical mappings is used when ease of schema
modification is paramount.

Fussell [9] suggests an extension to the RDBMS model in which objects (which
have identity and state) are allowed to be tuple attribute values. A tuple is defined as a truth
statement in the context of a relation. A tuple has attribute values which match the required
attributes in the relation and that state the condition that is known to be true. An example of
a tuple is: `<Person SSN#="123-45-6789" Name="Art Larsson" City="San Francisco">`. Tuples are values and two tuples are identical if their relation and attribute values are equal.
This merger allows objects in place of primitive or abstract data types. Therefore to
implement inheritance we use integrity constraints. When we specify that a domain is of a
particular type, we only allow objects that implement that type or any conformant subtype to
be values in that domain.

The following table [5], compares the three fundamental methods:

<table>
<thead>
<tr>
<th>Factors to consider</th>
<th>One table per hierarchy</th>
<th>One table per concrete class</th>
<th>One table per class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc reporting</td>
<td>Simple</td>
<td>Medium</td>
<td>Medium/Difficult</td>
</tr>
<tr>
<td>Ease of implementation</td>
<td>Simple</td>
<td>Medium</td>
<td>Difficult</td>
</tr>
<tr>
<td>Ease of data access</td>
<td>Simple</td>
<td>Simple</td>
<td>Medium/Simple</td>
</tr>
<tr>
<td>Coupling</td>
<td>Very high</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Speed of data access</td>
<td>Fast</td>
<td>Fast</td>
<td>Medium/Fast</td>
</tr>
<tr>
<td>Support for polymorphism</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 2.1 – Comparison between different mappings
2.4 Mapping Relationships

There are basically two ways of mapping object relationships in a RDBMS. The first is by using foreign keys only and the second is by using a relationship table. A foreign key is a data attribute that appears in one table that may be part of or is coincidental with the key of another table. Foreign keys relate a record in one table with a record in another and these can be used to map object relationships. A relationship table is a table whose sole purpose is to maintain the relationship between two or more tables. Its columns contain the foreign keys of the objects involved in the relationship.

In an object model, various relationships can exist between objects. These may be:

- 1 to 1 (husband – wife)
- 1 to many (mother – child)
- many to many (ancestor – child)
- ternary (or n-ary associations) (student – class – professor)
- qualified associations (company – office – person)

A qualified association is an association between two objects where the association is constrained or identified in some way. For example a Company can be associated with a Person through a position held by that Person. The position qualifies the association between the Company and the Person.

![Diagram](image)

**Figure 2.3** – Mapping relationships
Brown and Whitenack [8] suggest that the above relationships be mapped as follows:

- **Merge 1-to-1 associations with no special meaning into one of the tables.** If it has special meaning create a table based on the class derived from the association.
- **For 1-to-many associations, create a relationship table.** This is a table that consists of at least two columns, one which represents the primary key of the containing object, and the other the primary key of the contained objects. Each entry in the table shows a relationship between the contained object and the containing object.
- **A many-to-many relationship always maps to a relationship table that contains columns referenced by the foreign keys of the two objects.**
- **Ternary and n-ary associations should have their own table that reference the participating classes by foreign key.**
- **A qualified association should have its own table.**

Ambler [5] proposes that one-to-one and one-to-many relationships be mapped simply by including the key of one table in the other table (foreign key method). Figure 2.3 illustrates this mapping.

First, to map the one-to-one relationship between **Position** and **Employee**, the foreign key **PositionOID** is added to **Employee**. Alternatively, the foreign key **EmployeeOID** could have been added to **Position** instead. Second, the one-to-many relationship between **Employee** and **Task** is mapped similarly, with the foreign key **EmployeeOID** in **Task** because it is on the many side of the relationship. As for many-to-many relationships, it should be mapped using a relationship table.

In [8], a one-to-one relationship is mapped using foreign keys. Although a relationship table can also be used, it would not serve much purpose. As for one-to-many relationships, the mapping depends whether it is an aggregation or association relationship. For aggregation (part-of), it can only be implemented using an embedded foreign key column. Association (acquaintance) relationships can be implemented using either method—embedded foreign key or relationship table. Lastly, many-to-many relationships should be implemented using relationship tables.
2.5 Frameworks

The exact definition of a framework varies. Johnson [4] provides two common definitions:

"A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact."

"A framework is the skeleton of an application that can be customized by an application developer."

Larman [12] gives the following properties of a framework:

- Is a cohesive set of classes that collaborate to provide services for the core, unvarying part of a logical subsystem.
- Contains concrete (and especially) abstract classes that define interfaces to conform to, objects interactions to participate in, and other invariants.
- Usually (but not necessarily) requires the framework user to define subclasses of existing framework classes in order to make use of, customize, and extend the framework services.
- Has abstract classes that may contain both abstract and concrete methods.

Frameworks provides reuse at the largest granularity, which makes the task of application developers much easier, but for the framework developer it is not an easy task – designing a good framework is more difficult than an abstract class. Also frameworks tend to be application specific, e.g. the Microsoft Foundation Classes (MFC) which contain specialised classes to implement a graphical user interface in Windows.

Frameworks can be classified by their scope as follows: (Fayad & Schmidt [13])

- System infrastructure frameworks – these simplify the development of portable and efficient system infrastructure such as operating systems,
communications, user interfaces, and language processing tools.

- **Middleware integration frameworks** – these are commonly used to integrate distributed applications and components and are designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment.

- **Enterprise application frameworks** – these address broad application domains such as telecommunications, avionics, manufacturing, and financial engineering. They support the development of end-user applications and products directly and are the highest-level and most expensive to develop and/or purchase.

Also, regardless of their scope, frameworks can also be classified by the techniques used to extend them, which range along a continuum from white-box frameworks to black-box frameworks [13], [14].

A white-box framework is extended by defining subclasses inheriting from framework base classes and overriding pre-defined methods. Each method added by a subclass must abide by the internal conventions of its superclasses. Therefore it is ‘white-box’ because their implementation and internal structure must be understood to use them. Although white-box frameworks are widely used, they tend to produce systems that are tightly coupled to the specific details of the framework’s inheritance hierarchies.

On the other hand, black-box frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by defining components that conform to a particular interface and integrating these components into the framework. Therefore the user only needs to understand the external interface of the components, and as such the framework is ‘black-box’. Black-box frameworks are easier to learn to use but are less flexible since the internal structure cannot be modified.

One way to characterize the difference between white-box and black-box frameworks is to observe the accessibility of the state information of each instance [14]. In a white-box framework, this information is implicitly available to all the methods in
the framework. However, in a black-box framework, any information passed to constituents of the framework must be passed explicitly. Hence, a white-box framework relies on the intra-object scope to allow it to evolve without forcing it to subscribe to an explicit, rigid protocol that might constrain the design process prematurely.

A framework becomes more reusable as the relationship between its parts is defined in terms of a protocol instead of using inheritance. In fact, as the design of the system becomes better understood, the system should be evolved gradually towards black-box relationships [14].

2.6 Persistence Frameworks

When interoperating with a legacy database, a persistence framework will serve to bridge the gap between an object-oriented application and a relational database. A persistence framework is a reusable and extendable set of classes that provide services for persistent objects. Typically it has to translate objects to records and save them in a database, and vice-versa – to translate records to objects when retrieving from a database.

At the bare minimum, a persistence framework should provide the four basic CRUD (Create, Read, Update, and Delete) operations to store objects [16]. Although implementing this alone in a framework is workable, it would not suffice in a real-world application. More realistically, a persistence framework should have the following properties: [12]

- To store and retrieve objects in a persistent storage mechanism.
- Can commit and rollback transactions.
- Be extendable to support any kind of storage mechanism, such as relational databases, flat files, and so on.
- Require minimal modification to existing code.
- Be easy to use.
- Very transparent.

Ambler [6] gives in more detail the following real-world requirements for the design of a robust persistence framework:
• **Several types of persistence mechanism** – include support for flat files, relational, object-relational, hierarchical, network, and object databases.

• **Full encapsulation of the persistence mechanism(s)** – the internals of the mechanism must not be visible.

• **Multi-object actions** – support retrieval of many objects simultaneously.

• **Transactions** – support transactions, commit, and rollback operations.

• **Extensibility** – able to add new classes and databases.

• **Object identifiers** – use a unique object identifier for each object.

• ** Cursors** – support controlled object retrieval instead of returning a large number of objects at once to increase performance.

• **Proxies** – a proxy object is a reduced form of an object in which it only contains attributes necessary to identify the object. When the proxy object is selected only then is the real object retrieved from the database. This reduces traffic and increases performance.

• **Records** – support retrieval of records as well as objects from the database. Some legacy applications may still require records as input.

• **Multiple architectures** – support different network architectures, e.g. centralised, client/server, etc.

• **Various database versions and/or vendors** – support for various different databases.

• **Multiple connections** – support multiple, simultaneous connections to different persistence mechanisms.

• **Native and non-native drivers** – support various database drivers, e.g. ODBC, JDBC, and native drivers.

• **Structured query language (SQL) queries** – able to submit SQL queries to the database.

2.6.1 Design of a Persistence Framework

Based on the requirements above, there are various different designs that can be used in developing a persistence layer. There are fundamentally three methods of doing this: [16]
1. **Use an object layer** – Each domain object is subclassed from an abstract _PersistentObject_ class where each object would inherit the necessary CRUD functionality. This method is by far the simplest and easiest to implement. However, this means there is a high coupling between the objects and the database, which makes portability and scalability difficult.

2. **Use a broker** – This method involves the use of a broker, which is an entity that can read and write domain objects to and from a database. As such the broker needs to know the format of the domain objects beforehand, and performs the SQL necessary to read or write it. The advantage of this method is that the database-specific code is separated from the domain objects, making it portable and scalable. However it requires a significant amount of infrastructure to implement properly.

3. **Use data objects** – Each domain object is composed from a set of data objects that have a one-to-one mapping to the database tables. So when the domain object changes, the data object changes as well, and is saved

---

**Figure 2.4** – Design of an object layer persistence framework
when the domain object is saved. This alternative method is simple to implement and easy to understand, although it can be slow in performance and the one-to-one mapping may make it difficult to map complex object relationships.

The choice of method should be made based on the needed flexibility, scalability and maintainability.

Yoder et al. [16] presents a design based on the first method, as shown in Figure 2.4. Domain objects that need to be persisted are subclassed from PersistentObject which has the following attributes:

- **ObjectID** – this is a unique identifier for the object and can be the database key.
- **IsChanged** – identifies whether or not the object is ‘dirty’ which tells the persistence layer to write the object to the database.
- **IsPersisted** – identifies whether or not the object has ever been written to the database
- **owningObject** – identifies the parent object (in a complex object) and is used as a foreign key in the database.

The public methods for PersistentObject are as follows:

- **save** – writes the object to the database.
- **delete** – deletes an object data from the database.
- **load** – returns a single instance of a class with data from the database.
- **loadAll** – returns a collection of instances with all the data from the database for that particular class
- **loadAllLike** – returns a collection of instances of a class that match a particular criteria.

In addition, the PersistentObject is supported by various other services needed to perform its function. The Table Manager manages the mappings from an object to its database tables and columns. Connections to the database are maintained by the Connection Manager. The OID Manager is responsible for generating unique object
identifiers for every domain object that comes into existence. Lastly, the Type Converter handles the mappings between the object attributes and the database values and vice-versa. However, the user of the framework only needs to see the PersistentObject; everything else should be transparent.

Ambler [6] gives a different design, as in Figure 2.5, based on a combination of an object layer and a broker. Persistent domain objects also inherit from a PersistentObject, but they do not interface with the database directly, instead they communicate with a PersistenceBroker which handles the interface to the database. The classes have the following functions:

- **ClassMap** – A collection of classes that encapsulate the behaviour needed to map classes to relational tables.
- **Cursor** – This class encapsulates the concept of a database cursor, which
allow retrieval of subsets of information from the persistence mechanism at a time.

- **PersistenceBroker** – Maintains connections to persistence mechanisms, such as relational databases and flat files, and handles communication between the object application and the persistence mechanisms.

- **PersistentCriteria** – This class hierarchy encapsulates the behaviour needed to retrieve, update, or delete collections of objects based on defined criteria.

- **PersistenceMechanism** – A class hierarchy that encapsulates access to flat files, relational databases, and object-relational databases.

- **PersistentObject** – This class encapsulates the behaviour needed to make single instances persistent and is inherited by domain objects that require persistence.

- **PersistentTransaction** – This class encapsulates the behaviour needed to support transactions, both flat and nested, in persistence mechanisms.

- **SqlStatement** – This class hierarchy knows how to build insert, update, delete and select SQL statements based on information encapsulated by ClassMap objects.

Most of these classes are not visible to the user of the framework. Only the PersistentObject, PersistentCriteria, PersistentTransaction, and Cursor classes need to be known to make use of the framework.

### 2.6.2 Existing Persistence Framework Products

There are already various commercial products that offer persistence services for objects. The majority provide interfaces in programming languages such as C++ and Java, and supports technologies such as CORBA (Common Object Request Broker Architecture), EJB (Enterprise Java Beans), and ODBC (Open Database Connectivity). Most of the products utilise an object layer or broker design, as discussed in the previous section. This section will summarize a few of the many products that are available in the market.

CocoBase [17] is a mapping framework for rapid application development in Java. It maps data sources such as relational, mainframe and object databases as well as
transport technologies such as CORBA, EJB, and JDBC drivers into a single application programming interface.

Java Blend [19] by Sun Microsystems and DbGen [18] by 2Link Consulting are object-relational mapping tools in Java. They generate data-aware Java objects in source code form so that the programmer can use ordinary Java methods to insert, update, delete and query the database without knowing any SQL.

The Secant Extreme POS [20] provides persistence services for C++ applications. It has three major layers, the persistent object manager, persistent data service, and the data store framework, from top to bottom respectively. It also uses an object workspace which is an object cache to ensure that an object is only read from the database once during the course of the transaction.

For Smalltalk applications, ODBTalk [21] provides a database framework for accessing ODBC data sources. It also includes a persistence framework for the mapping of Smalltalk objects to relational databases. Its design consists of several subsystems: type, object, broker, class generation, and forms generation.

OAdapter is Hewlett-Packard's object/relational adapter which enables object-oriented developers to share a common object model stored in the ORACLE7 relational database management system (RDBMS). The OAdapter language can be combined with functions implemented in C++, Smalltalk or C.

2.7 Application in the Healthcare Industry

The healthcare industry, which includes hospitals, clinics, healthcare organizations, has not been quick to pick up on information technologies. Many patient record systems are still paper-based and even most computerised systems are based on legacy technology and not completely paper-free. This is partly due to the difficulty in storing patient records such as X-rays, MRI and CAT scans, charts, sound, and video.

According to Peter Waegemann of the Medical Records Institute [22], there are five stages in implementing an electronic patient record, from a simple automated medical record system to a complete electronic health record, The majority of health
institutions are only at stage one and two, which means that they are just starting off on an automated system parallel to a paper-based system and at most have implemented a completely electronic record storing the patient’s details. At the other end, stage five, is a universal health record which stores complete information about a person’s health which can be accessed from anywhere, subject to permission granted by the person. No such system has been implemented to date.

Adoption of object-oriented technology in this area is rather scarce, primarily due to its infancy and the difficulty in migrating. Therefore as legacy systems are upgraded, there will be a need for a transition period in which the new object-oriented system will co-exist with the legacy system. In a recent survey conducted by the Medical Records Institute [23], healthcare industry professionals were asked how they would deal with their legacy systems. The survey involved 477 respondents, mainly from the United States. The responses are shown in the following table.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace legacy systems selectively over time as part of a multi-year migration strategy.</td>
<td>38%</td>
</tr>
<tr>
<td>Retain all or most of the legacy systems and arrange to have the clinical data repository or electronic health record (EHR) interface or network to them.</td>
<td>37%</td>
</tr>
<tr>
<td>Replace all or most legacy systems because the clinical data repository or EHR will be part of a comprehensive healthcare information system that includes applications that replace the legacy systems.</td>
<td>16%</td>
</tr>
<tr>
<td>Replace all or most legacy systems because they are unable to interface to the new stand-alone clinical data repository or EHR.</td>
<td>9%</td>
</tr>
</tbody>
</table>

Table 2.2 – Survey of solutions to the legacy system problem

From the results of the survey, the two most preferred solutions (75% of all respondents) are to implement a gradual migration strategy or to have the new system co-exist with the old one. This would require an intermediate layer to interface the two systems and thus there will be demand for a persistence framework to interface object-oriented systems to legacy databases. A generic persistence framework would suffice for this purpose, provided it supports the common database systems used in the healthcare industry.
The other two solutions are to replace the legacy system completely, which accounts for only 25% of the respondents. This can be mainly attributed to the cost and difficulty in implementing such a strategy. Small organizations, such as clinics, would be more successful in this approach than large hospitals.