How should one model persistent objects and relationships between them in an object-oriented system? This question often arises in application domains where information systems play a central role. As argued in chapter 2, a strong case can be made against entity–relationship modeling and its variations in object-oriented contexts, since it breaks the inherent seamlessness of the approach. However, even if object-oriented technology is now rapidly moving towards commercial acceptance on a broad scale, relational databases will most likely continue to play an important role as data repositories for a long time yet, including many object-oriented applications.

There are several reasons for this. First of all, statistics have proven that the average lifetime of stored data is far greater than the average lifetime of applications handling the data. Thus while applications are being modified and replaced, corporate data, although extended and updated, tends to remain where it sits. Therefore, many information systems have grown extremely large and the cost of a complete data conversion may not always be justifiable.

Moreover, databases are often accessed and manipulated by many different applications in heterogeneous environments (often geographically distributed), and it may not be worthwhile to rewrite all of these applications to comply with a different database organization. Other reasons may have to do with company policies, previous investment in database software and expertise, performance requirements (transaction processing, concurrent updates, average uptime), and data security (consistency controls, recovery/rollback, authorization).

The conclusion is that bridges are often needed between the relational and object-oriented worlds. The purpose of this last case study is to discuss how object models and relational models can be made to coexist in a system. The approaches illustrated are drawn from actual working implementations, but since a full discussion could easily fill a book of its own, they have been considerably simplified.
11.1 FROM DATA STORAGE TO OBJECT PERSISTENCE

Let us first recall what is needed by an object-oriented execution model. Object-oriented applications handling massive quantities of data often end up using and creating large numbers of objects. Most existing object-oriented environments still run on top of operating systems that do not support the basic run-time requirements of an object-oriented approach: built-in object allocation, automatic reclamation of unreachable objects (garbage collection) regardless of physical location, and transparent paging at the object level of both transient and persistent objects. Until basic facilities like these become widely available as part of standardized families of operating systems, each object-oriented environment needs to implement all or part of them as a separate virtual machine (run-time system).

Not to burden applications with implementation details, object-oriented environments must offer powerful means to handle both transient and persistent objects. Large quantities of transient objects can often be taken care of by traditional garbage collection in combination with virtual memory management at the operating system level. Small amounts of persistent objects, in turn, can be encapsulated by the run-time system using database library classes interacting with an ordinary file system. However, to handle a potentially very large number of persistent objects, more complete database capabilities are usually required. A mapping is needed whenever object-oriented applications interface with legacy systems or with databases that do not interoperate with an object-oriented environment.

Interoperability in this context means more than just a basic coupling. Owing to the high level of object integration required (object type and format, polymorphism) distribution transparency of objects with implicit access via the development environment is rapidly becoming an important issue. Some technical approaches addressing the problems involved are beginning to emerge, whose aim is to support the notion of an “object bus” and connect heterogeneous object-oriented applications directly at the object level [OMG 1991].

Seamlessness between execution model and persistent data means that values and types directly map the class instances used by the execution model; there is no “impedance mismatch” between primary memory and disk-resident data. An ideal solution completely frees client applications from storage details and permits virtual addressing of an infinite object space. In such cases, the object-oriented run-time system transparently pages in or out clusters of objects according to their status and behavior in the application: frequency of access, reachability, expected lifetime, and so forth.

Consider the following program fragment from a developer’s standpoint. The declaration and qualified call translate at execution time into: “apply routine
register defined and exported by class CUSTOMER to an instance of CUSTOMER (or one of its descendant classes) attached to an attribute called attendee.

\[
\text{attendee: CUSTOMER} \\
:\text{attendee.register}
\]

In a fully transparent persistent environment, the above is complemented by: “regardless of the effective location of the object referred to by attendee at the time the call is executed.”

An object is persistent if its existence is independent of the system session in which it was created. A persistent object continues to exist until it either becomes unreachable or is explicitly deleted. Various techniques can be employed in object-oriented environments to make an object persistent. Figure 11.1 depicts some possibilities.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>customer := new persistent CLIENT</td>
</tr>
<tr>
<td>2</td>
<td>customer: persistent CUSTOMER</td>
</tr>
</tbody>
</table>
| 3 | persistent class CUSTOMER \\
|   | : |
|   | end |
|   | customer: CUSTOMER |
| 4 | class CUSTOMER \\
|   | inherit PERSISTENT \\
|   | : |
|   | end |
|   | customer: CUSTOMER |
| 5 | persistent_collection: PERSISTENT_UNIVERSE \\
|   | customer: CUSTOMER \\
|   | : |
|   | persistent_collection.put (customer) |

**Figure 11.1** Various persistency schemes

The first three examples in figure 11.1 introduce specific language constructs: (1) extended object creation mechanism, (2) extended type declaration of entities referring to objects, and (3) extended class declaration mechanism. The last two examples use predefined classes to achieve persistency: (4) all children to a common ancestor become persistent, and (5) a persistent object container accepts any object reference, and all objects inserted into the container automatically become persistent.
Regardless of the specific mechanism used, we may adopt the following *deep persistency* principle: all objects reachable through successive references from a persistent object also become persistent. This ensures consistency of the system state (class invariants). Unless the transitive closure of objects referred to by a persistent object is also stored, some objects may become invalid.

Objects explicitly made persistent through some scheme like the ones in figure 11.1 are sometimes called *persistent roots* (not to be confused with root objects starting up system executions). All other objects may dynamically be or not be persistent, depending on whether they can be reached from a persistent root or not.

Persistency in BON is defined as a class property and persistent classes can be marked as such by a special class header annotation (bullet). This is often of interest during analysis, since figuring out what objects need to survive system sessions may be a good way to increase problem understanding. However, it would be too restrictive to require that only persistent objects can be instantiated from a class marked as persistent.

There may be situations in a system where a temporary object needs to behave exactly like a persistent one, and forcing the creation of two nearly identical classes in such cases does not make much sense. Therefore, marking a class as persistent in BON means that its objects are *potentially* persistent.

We conclude this section by stating two principles regarding persistency, which are important for the seamlessness and reversibility of the BON approach. The aim is to keep analysis and design models simple and consistent, independently of where the objects will ultimately reside.

**Principle 1**

There should be no syntactic distinction between persistent and transient data in terms of how they are defined and used.

**Principle 2**

The persistent object model should be designed as a seamless part of the full static model.

With these preliminaries we are ready to take a look at the problems involved when object persistency (or part of it) is to be based on an underlying relational model. We will discuss an approach for achieving a high degree of transparency with regard to object retrieval and update—in spite of the structural differences between object models and relational databases. The focus will be on the dynamic construction of queries to reduce as much as possible the static dependency of applications on the actual database schema.
11.2 OBJECT MODELS AND RELATIONAL MODELS

Our objective is to design an integration layer that can access data in different storage formats and automatically convert between them. This layer may also have its own local repository, which may be used for caching to avoid constant data transfer through a gateway. The kind of environment envisioned is illustrated in figure 11.2.

![Diagram of transparent integration of heterogeneous storage]

**Figure 11.2** Transparent integration of heterogeneous storage

Different forms of coupling are possible. At the highest level of integration, persistent class instances are stored along with their features and class descriptions. With a more pragmatic approach, only the data part (state variables) of persistent objects is stored. In both cases, the database needs to be closely integrated with the execution model.

However, when the logical structure of the available persistent storage is totally unrelated to the object model (relational databases, flat files, indexed files) a separate interface layer is needed to do the necessary transformations. In figure 11.2 a SQL interpreter is used as backend to retrieve and store data in a relational database.18

In either case, it should be possible to define application object models, where the persistency decisions are kept completely free from any implementation choice, and all application objects are accessed the same way whether transparently constructed from external data or not.

---

18 The relational data language SQL was earlier named SEQUEL and is usually pronounced as though it still were. We therefore write “a SQL…” rather than “an SQL…”.\(^2\)
Integrity constraints

A number of data integrity rules are usually enforced in a relational system to prevent certain types of inconsistencies from entering the database. These rules, commonly known as integrity constraints, address various aspects of the semantic content of stored data. We will mention a few of them below, and see how they translate to a BON object model.

Domain integrity refers to type checking between the values used in query expressions and the declared types of the corresponding entities. All RDBMS provide the necessary level of checking to avoid any violation of the type system rules. Since BON is statically typed, it is assumed that the supporting environment (CASE tool at the analysis and design level, and programming system at the implementation level) will detect any type error.

Referential integrity has to do with the consistency of references between schema elements. Whenever an entry in a relational table refers by a foreign key value to an entry in another table, that other table must exist and have an entry with matching primary key value. Any modification of the database content must keep all related tables consistent and prevent the introduction of unmatched references. These checks are usually supported at the RDBMS level. It is assumed in BON that referential integrity is captured by assertions in class descriptions. In the example given in figure 11.3, integrity is guaranteed by the postconditions associated with the routines bid_farewell and retire.

User-defined integrity is usually taken care of by stored procedures or triggers in relational databases. With BON it is part of the object-oriented data description. The creation routines of a class are responsible for ensuring that each object of the class is created in a consistent state. Any operation changing the state of the object will, if necessary, trigger other monitoring operations to ensure that the consistency is maintained.

At execution time, depending on the supporting environment, integrity violations may invoke exception handlers, rollback procedures, and other recovery mechanisms.
We now turn to the design of an integration layer coupling an object model with a SQL gateway. There are three aspects of such a layer that need to be addressed:

- How to retrieve data from an existing relational database.
- How to regenerate application objects from the retrieved data.
- How to design a relational schema suitable to store and retrieve a given persistent object model.

Our aim is to find a generic model to tackle the problems, which is as independent as possible of both the object-oriented application and the relational database system used.

11.3 A RELATIONAL DATABASE WRAPPER

Short overview of the relational model

In relational databases, information is modeled as relations defined on finite sets of data values called domains. A relation is a set of ordered lists of data values called tuples.

Each tuple is an element of the cartesian product of domains \( D_1 \times D_2 \times D_3 \ldots \) (set of all possible ordered lists of values with one element drawn from each domain). Each occurrence of a domain in the definition of a relation is called an attribute, and the same domain may occur several times. Note the difference between domain and attribute: a domain is a basic pool of permissible values, while an attribute represents the use of a domain within a relation.

Domains and relations are implemented as tables with \( m \) rows and \( n \) columns. Each row corresponds to a tuple (an element in the set), and each column to a relational attribute. For this reason, RDBMS vendors usually use the terms table, column, and row instead of relation, attribute, and tuple.

The great majority of relational systems are normalized, which means that all domain elements must be atomic values. All values in a given domain have the same type chosen from a small set of predefined basic types, such as: INTEGER, FLOAT, DOUBLE, DATE, CHAR, STRING, MONEY. Each attribute (or column) of a relation has a name, so it can be referred to without using its relative position, and a type, which is the basic type of the corresponding domain.

Relations are usually defined with constraints imposed on the tuples to avoid data duplication or cross-dependency. A common constraint on a relation is to require that each tuple be uniquely identifiable by a subset of the attribute values. Such a subset is called a primary key. Often one attribute is enough to identify
tuples, in which case we have a single-attribute primary key.

Access to relational data is achieved through general set operations: selection, projection, product, join, union, intersection, and difference. In addition to tables predefined in the relational schema, new tables may be created dynamically through such operations. The data access set operations are expressed in a relational database language called SQL (Structured Query Language) used to store, retrieve, and modify information in the database. (SQL has become the de facto standard in the relational database world, and was accepted as an international standard by ISO in 1987. The latest ISO version, SQL/92, became ratified in 1992 [Date 1993].)

In figure 11.4, three example tables are shown: CUSTOMER, INVOICE, and PRODUCT. The header of each table shows the table name and the names of each attribute. Below the double line are the tuples, whose values conform to the basic type of each attribute (these types are not shown in the table).

The most frequently exercised operations in relational database applications are usually simple selections of tuples whose attribute values satisfy certain conditions, insertion or deletion of tuples, and change of attribute values in

| CUSTOMER | | | | |
|---|---|---|---|
| Client_Id | Name | Address | Zip_Code |
| A45 | Jack’s Snack | 899 Ventura Blvd, La Cienaga | CA 92340 |
| L20 | Red Lobster | 9B Nathaniel Hawthorne, Tauton | MA 02780 |

*Primary key: (Client_id)*

<table>
<thead>
<tr>
<th>INVOICE</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase_Order</td>
<td>Product_number</td>
<td>Qty</td>
<td>Client_Id</td>
</tr>
<tr>
<td>940120-010</td>
<td>1022</td>
<td>500</td>
<td>A45</td>
</tr>
<tr>
<td>940322-093</td>
<td>1024</td>
<td>80</td>
<td>Y89</td>
</tr>
</tbody>
</table>

*Primary key: (Purchase_Order, Product_number)*

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Product_number</td>
<td>Description</td>
<td>Unit_Price</td>
</tr>
<tr>
<td>1022</td>
<td>Corned Beef 1.54 oz can</td>
<td>0.99</td>
</tr>
<tr>
<td>1023</td>
<td>Snails in garlic butter 0.8 oz bag</td>
<td>4.99</td>
</tr>
<tr>
<td>1024</td>
<td>Peeled tomatoes 1.9 oz bottle</td>
<td>1.99</td>
</tr>
</tbody>
</table>

*Primary key: Product_number*

---

Figure 11.4  Tables from a relational schema
tuples. Selection is mostly combined with projection, which means that only a subset of the attribute values are retrieved.

The join operation is important for more complex retrieval. For example, assume we want a list of all customers who ordered products with a unit price of at least five dollars. The result should be presented as a table with the following attributes: client id, client name, client address, product description.

The combination of these attributes does not exist as a table per se in our schema, but it is possible to join our three tables to obtain the requested information. Using SQL syntax, the selection can be expressed as follows:

\[
\text{select CLIENT.client_id, name, address, description from CLIENT, INVOICE, PRODUCT} \\
\text{where CLIENT.client_id = INVOICE.client_id and} \\
\text{INVOICE.product_number = PRODUCT.product_number and} \\
\text{PRODUCT.unit_price >= 5.0}
\]

The result of a selection query is generally a set. Therefore, SQL provides facilities to iteratively fetch each matching row. A cursor maintained by the database server points to the currently retrievable row, and standard operations can be used to move the cursor from one row to another within the result.

The relational model for database management, originated in the late 1960s by E. F. Codd, has a strong mathematical foundation [Codd 1970]. It has been thoroughly researched and a large number of rules and criteria for relational data organization and manipulation have been proposed [Codd 1985a, Codd 1985b, Codd 1990]. For good comprehensive overviews of the area, see [Date 1990, Date 1983, Date 1993].

**Designing a database interface cluster**

Any cluster layer interfacing a relational database and an object-oriented system would be responsible for managing server sessions, maintaining the relational schema, performing queries and updates, and doing the mapping between rows and objects. To summarize this, let us define the major abstractions of such a layer and group them as shown in the cluster chart of figure 11.5.

We can also display the classes in a first static architecture sketch as shown in figure 11.6. The implementation of each model class will encapsulate a set of external calls to the database server. The DATABASE_INTERFACE cluster is a client of class ANY because any type of object may become persistent and thus need to receive external data.

Our first aim is to outline the interface of a general reusable cluster for accessing a relational database. The specification of this cluster is at a rather technical level, so we will skip the class chart definitions and move directly to the formal class descriptions.
### Purpose
Layer to make relational database manipulations transparent.

### Indexing
**Keywords:** object and relational coexistence, rdbms interface

<table>
<thead>
<tr>
<th>Class/(Cluster)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB_SESSION</td>
<td>Session manager responsible for handling the connection to the database server and for tracking the proper or non-proper completion of all transactions.</td>
</tr>
<tr>
<td>DB_QUERY</td>
<td>SQL wrapper sending selection commands to the database server.</td>
</tr>
<tr>
<td>DB_CHANGE</td>
<td>SQL wrapper sending store, update, and delete commands to the database server.</td>
</tr>
<tr>
<td>DB_RESULT</td>
<td>Representation of one matching row returned by the database server in response to a SQL selection.</td>
</tr>
</tbody>
</table>

**Figure 11.5** First candidate classes in interface layer

**Figure 11.6** First cluster sketch

Class `DB_SESSION` encapsulates the most important primitives to handle transactions between an application and the database server. Its interface is outlined in figure 11.7.

Class `DB_QUERY` sends SQL selection queries to the database server and stores the resulting table rows. Clients can then iterate on the result supplying a callback routine to process the table rows, one by one. The interface description of class `DB_QUERY` is shown in figure 11.8.

Typically, the callback object will be the client object itself, inheriting from `ACTION` and defining the `execute` feature. Since the client already has a reference to the `DB_QUERY` object on which the iteration was invoked, the `execute` routine will be able to access the database cursor of the corresponding selection.
Class `DB_RESULT` represents the database cursor pointing to the current table row returned by the database server. It is responsible for the conversion of data fields from the SQL structure on the server side into corresponding basic object attributes that may be accessed in a normal way by the object model. Any of the fields can thus be inspected, which gives clients full control to do whatever processing is needed.

However, in many cases the main part of the action for each returned row will be to transfer some or all of the data fields into the corresponding attributes of some result object. Therefore, the `load_object` command of `DB_RESULT` (see figure 11.8) will automatically convert and load data from the fields of the current row into an object supplied as an argument.

Each `basic` attribute of the argument object that has a name and a type which matches an attribute of the table row will be set to the corresponding value. An object attribute is considered basic if its type corresponds directly to a type defined in the relational database. The mapping from one type system to another can be preset in a utility class and accessed when needed.

It is the client’s responsibility to ensure that each basic object attribute which is to receive a value corresponds exactly (by name and type) to one of the table

---

<table>
<thead>
<tr>
<th><strong>DB_SESSION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connect</code></td>
</tr>
<tr>
<td>&quot;Connect application to database server.&quot;</td>
</tr>
<tr>
<td><code>is_connected</code></td>
</tr>
<tr>
<td><code>disconnect</code></td>
</tr>
<tr>
<td>&quot;Disconnect application from database server.&quot;</td>
</tr>
<tr>
<td><code>¬is_connected</code></td>
</tr>
<tr>
<td><code>commit</code></td>
</tr>
<tr>
<td>&quot;Update database with last modifications.&quot;</td>
</tr>
<tr>
<td><code>is_connected</code></td>
</tr>
<tr>
<td><code>rollback</code></td>
</tr>
<tr>
<td>&quot;Backup to previous state.&quot;</td>
</tr>
<tr>
<td><code>is_connected</code></td>
</tr>
</tbody>
</table>

`is_connected`: `BOOLEAN`  
"Is application connected to the database server?"

`transaction_status`: `VALUE`  
"Status of last performed transaction"
row attributes. Furthermore, the names and types of the attributes of an object must be dynamically accessible at execution time for the automatic data transfer to work. In object-oriented environments where this information is not available, a corresponding table (preferably generated from the corresponding class description by some tool) may have to be attached to each persistent class.

Note that the automatic loading does not require all basic attributes of the receiving object to match columns in the table row, nor all columns to correspond to an object attribute. Only basic attributes with matching name and type will be transferred. This convention has two advantages:
Some basic object attributes may be left out of a query if they are considered uninteresting in some context (perhaps given default values).

Several objects of different type may be loaded, one at a time, from the same query result. This will be important for our design of the higher-level layers.

Class `DB_CHANGE`, finally, is simply used to pass a SQL statement requesting an update, deletion, or insertion in the relational database.

**Mapping non-basic object attributes**

So far, we have only discussed storage and retrieval of basic object attributes. Usually, however, the attributes of most objects in an object model will be a mixture of basic and non-basic types. If we look at the class `REGISTRATION` from the conference case study, whose interface is repeated in figure 11.9, we find two attributes that relate to other model classes, namely `PERSON` and `TUTORIAL`.

```
REGISTRATION

attendee: PERSON
registered_at: DATE
amount_paid: VALUE
invoice_sent: BOOLEAN
confirmed: BOOLEAN
paper_sessions: BOOLEAN
selected_tutorials: SET[TUTORIAL]
```

Figure 11.9 A persistent class description

These attributes represent object references which cannot be mapped directly to relational attributes. (In this case study, we will assume that all typed features of persistent model objects represent class attributes rather than functions, unless otherwise stated.)

However, we can use the automatic data transfer previously described to have the basic attributes of a `REGISTRATION` object (`registered_at`, `amount_paid`, `invoice_sent`, `confirmed`, and `paper_sessions`)) automatically initialized from the database rows. In an eventual implementation, general types such as `VALUE` in figure 11.9 will have been specialized to a basic type of the programming language making the correspondence to the relational types clear. A possible
mapping could be the one in figure 11.10.

The \textit{PERSON} and \textit{TUTORIAL} objects corresponding to \textit{attendee} and \textit{selected_tutorials} can then be initialized separately with rows coming from other relations, and the corresponding references filled in by the application object responsible for restoring \textit{REGISTRATION} objects.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTERED_AT</td>
<td>DATE (MM/DD/YY, HH24:MI:SS)</td>
</tr>
<tr>
<td>AMOUNT_PAID</td>
<td>FLOAT</td>
</tr>
<tr>
<td>INVOICE_SENT</td>
<td>CHAR (1)</td>
</tr>
<tr>
<td>CONFIRMED</td>
<td>CHAR (1)</td>
</tr>
<tr>
<td>PAPER_SESSIONS</td>
<td>CHAR (1)</td>
</tr>
</tbody>
</table>

\textbf{Figure 11.10} Relational table representing class \textit{REGISTRATION}

When a persistent object structure is stored in the database, we must capture the unique identity of each object through primary keys in the database tables. Sometimes basic attributes already exist, whose values can be used to fully identify each object. In our case, it might be possible to use \textit{registered_at} as the primary key, provided that time information of enough granularity is included and that registrations are not entered simultaneously.

If such an attribute (or group of attributes) does not exist, we need to add an extra field containing a unique identifier. Such an extra field may also be useful for efficiency purposes when the existing primary key is long and many other persistent objects will refer to this one.

Object reference attributes may be stored as fields pointing to other tables representing the referenced objects (so-called foreign keys). Different strategies may be used to represent references between model objects, depending on the corresponding instance multiplicity.

- If the reference is \textit{one-to-one} (as for \textit{attendee} in class \textit{REGISTRATION} above), an extra field in either one of the tables \textit{REGISTRATION} or \textit{PERSON} containing a primary key to the other is enough.

- If the reference is \textit{one-to-many} (as for \textit{children} in a class \textit{MOTHER}), an extra field in the table representing the “many” pointing to the table representing the “one” will do.

- If the reference is \textit{many-to-many} (as for \textit{selected_tutorials} above), we may need to represent the relation as a joint table \textit{TUTORIAL_ATTENDANCE} containing two primary keys: one pointing to table \textit{REGISTRATION} and one to \textit{TUTORIAL}.

However, if the maximum number of instances referred to by either side is low, another possibility is to map the reference into a fixed number of extra fields containing either a zero reference or a primary key value.
For example, if the maximum number of tutorials each person may attend is four, a separate table can be avoided by adding attributes tutorial1, tutorial2, tutorial3, and tutorial4 to table REGISTRATION.

There are also various strategies for mapping inheritance relations to relational database schemas. For overviews, see [Rahayo 1993, Premerlani 1994, Blaha 1994].

A scenario

We conclude this section with a scenario illustrating how an application may use the query facility with automatic data transfer. The dynamic diagram is shown in figure 11.11.

**Scenario: Query database and load result**

1–3  A client object starts a database session, creates a query object, and invokes a SQL query on it.

4–7  The client creates a new registration and starts an iteration on the query result, supplying itself as action object. The query object resets the cursor to point to the next resulting row, and invokes the client action routine.

8–10 The action routine obtains the cursor object and tells it to load the registration object from the table row.

**Figure 11.11**  Scenario showing access and automatic load

### 11.4 INTERFACING AN EXISTING RELATIONAL SCHEMA

Another important issue for the database encapsulation is the structure of the relational schema (what precise tables and attributes we need to access). Various factors affect the design of such a schema.
Factors determining the schema

Basically, there are two situations regarding choice of data organization:

- The relational schema mapping the object model will not be used for any other purpose.

- The relational schema is also used by other applications that do not necessarily have exactly the same view of the world.

In the first case, the database can be used as an implementation vehicle with relational tables replacing flat files, indexed files, or any other data storage facility. This situation leaves a great deal of freedom in designing the relational schema, and data administrators may capture the object model in a way that best fits the application. The goal is then to find a suitable tradeoff between a mapping giving good performance and one that is easy to understand and maintain.

In the second case, many additional factors must be taken into account. Often the schema and associated database (or a significant part of it) is a legacy to our system, which we may not be able to do anything about. The way attributes are distributed in the object model may then be very different from the organization of the corresponding data on the relational side.

This may result in SQL queries performing complicated joins across numerous scattered tables, which may degrade performance. Often the solution is to perform a number of pre-joins on selected tables and store them temporarily during an object-oriented session. The initial cost at session start is then offset by the improved performance during data transfer between the models.

With a growing number of heterogeneous applications sharing persistent information across networks, the ability to adapt to existing structures is important. In fact, even if the schema can be optimally tailored to an object model (no initial legacy), the database organization tends to become much more rigid once it has been filled with large amounts of data.

As time passes, the object model will gradually change (perhaps even faster than with traditional models, because of the inherent flexibility of the object-oriented approach). The impedance mismatch may then make it too expensive to continually modify the relational structures to keep up with the changes. This means we may have to face the legacy situation soon enough, even in cases where the only applications ever accessing the data are object oriented!

The rest of this case study will be devoted to a detailed discussion of how existing schemas can be mapped to model objects. We will design a set of clusters of reusable classes enabling applications to become independent of the exact database organization, and show how this cluster fits in with the general database encapsulation presented in the previous section.
We will see how the design can be done gradually in layers raising the level of abstraction to overcome the structural differences between the relational and object models.

**Schema dependency**

If a typed implementation language is used, making static changes in the object model implies recompilation of the application. This is reasonable, and usually corresponds to a new version of the software. However, updates of the relational schema in a database shared by many applications may occur frequently (new columns added to tables, new tables added, minor reorganizations for efficiency). A solution which forces recompilation and reinstallation of an application each time a schema change occurs is therefore too rigid in most cases.

For this reason, we should strive to keep our applications free from exact knowledge of the mapping between the object model and the relational database. Rather than placing complete information directly in the static class structure about the names of each database column accessed and the table it resides in, the mapping should be dynamically reconfigurable by modification of data in some repository. But how can we obtain adequate performance without integrating the relational structure in our object-oriented applications?

**A virtual database**

One solution is to define a virtual database containing a set of virtual tables, and then make applications statically dependent only on this database. (Such virtual tables are known as views in RDBMS terminology.) If the virtual database is chosen reasonably close to the real database, the conversion between the two schemas will be straightforward, and can be effected by SQL statements dynamically maintained as stored procedures or persistent strings. This gives freedom to rename and create new tables in the database and to rename and move around columns between them without changing the static structure of the applications.

Regarding the logical representation of each column in the database, the amount of freedom depends of course on the complexity of the mapping. If the database stores temperature in degrees Fahrenheit and the object model uses the Celsius scale, we cannot expect SQL to hide this fact. Also, even if the SQL dialect provided by the database server would allow expressions to retrieve two database columns given_names and last_name and directly return a concatenated attribute name to the object model, it will hardly be possible to do the reverse on update.

Therefore, the logical structure of the real database must usually be mirrored by the virtual database, and applications must be statically dependent on the
representation form chosen for each column interacted with. However, independence of the exact tables in which the columns reside, as well as of any table or column renaming, is still a great advantage.

Each virtual table is represented by a class encapsulating a set of basic attributes. The virtual table classes (which collectively represent the database visible to the application) will be named row classes. Instances of row classes are called row objects and each row object will act as a gateway to the real database.

Each virtual table is chosen so that there is a simple mapping between its attributes and the attributes of the real database tables. The persistency classes will encapsulate operations to do the conversion using a SQL database server. However, the corresponding SQL statements will be maintained outside the application to always reflect the current state of the real database schema. An application can load the correct SQL mapping at startup time or, in case it needs to run continuously for a long time, be triggered to reload any change that may occur during execution.

**An example application**

To illustrate the above approach, let us select four of the persistent classes from the conference case study (chapter 9). The corresponding class descriptions, showing only the features which we assume will be implemented as attributes, are repeated in figure 11.12.

We also assume there is a corporate database which is to be used for mapping relevant parts of our object model. In the corporate database, we find four tables containing information that can be used to represent the basic attributes of classes PERSON and REGISTRATION. These are shown in figure 11.13. There are no existing tables corresponding to classes TUTORIAL or PRESENTATION.

The CUSTOMER and AFFILIATION tables come from the company’s general customer register and the INVOICE table from its accounting system. The REGISTRATION table is assumed to have been designed as part of an older system which handles conference registrations but not the technical program. We also assume that at present there is nothing we can do to change the formats of these tables. This represents a kind of legacy situation not uncommon in practice.

Since some objects are more difficult than others to map to a relational system, it may be an advantage to have a relational and object-oriented persistency mix. In this case, we choose to store and retrieve PERSON and REGISTRATION objects in the relational database, while TUTORIAL objects will be stored using some object persistency mechanism provided in the language environment.
Figure 11.12  Simplified persistent object model

This strategy also fits well with general performance considerations. Since the tutorial objects are relatively few and frequently accessed, they should remain in main memory during system execution. Person and registration objects, on the other hand, may occur in great numbers but only the ones currently being processed need fast access.

The general access, manipulation, and update of our object model and corresponding relational data can now be outlined as follows:

- **Creation of new tutorial objects.**
  These objects must be present before any registrations can be accepted, since choice of tutorials is part of the registration data. Before the reference to `speakers` and `authors` is filled in, the CUSTOMER and AFFILIATION tables are searched to check whether some of the persons
are already present in the corporate database. If this is the case, all attribute values of the PERSON objects are initialized with the corresponding values from the database. Persons not found in the database will be created and initialized from the input data on the object model side.

- **Creation of new registration objects.** These objects are created from registration input data and will refer to the already defined tutorial objects. As above, assigning the reference attendee will either retrieve an old PERSON object from the database, or create a new object.

- **Update relational database.** Database updates may be performed at regular intervals, or when requested by an operator. Unless some personal data needs to be
corrected, the CUSTOMER and AFFILIATION tables will only be updated if new persons have been entered in the object model. The REGISTRATION table will be updated for each new registration and when existing ones are modified. The latter occurs, for example, when a letter of confirmation has been sent or a tutorial selection is changed.

A virtual database interface

We are now in a position to start putting things together and sketch a general design for mapping persistent objects to relational systems whose tables are not in direct accordance with the object structure. We will use the architecture depicted in figure 11.14.

![Figure 11.14 Persistent object management (outline)](image)

The two row classes of the virtual database are defined in figure 11.15. They are the virtual relational representation of the corresponding persistent objects. A row class encapsulates the interface of the RDB cluster. It may be given a SQL selection, in which case it will create a DB_QUERY object, attach itself to it, and forward the query.
The `set_action` and `iterate_on_result` commands can then be used to scan through the resulting table rows and `load_from_cursor` will load the current row object, so it can be further processed. The resulting rows must of course contain all columns corresponding to the attributes of the row object, so it can be loaded. This is the responsibility of the client passing the SQL query.

Each persistent class will have two other classes corresponding to it: a `row` class representing the object in the virtual database, and a `manager` class to do the conversion between the representations. Each manager will only know the
Comparing the columns of figure 11.13 with the row classes shows that the attributes of \texttt{PERSON\_ROW} correspond directly to a subset of the columns of \texttt{CUSTOMER} and \texttt{AFFILIATION}, which can easily be obtained by join and projection using SQL. The same is true for \texttt{REGISTRATION\_ROW} with respect to tables \texttt{REGISTRATION} and \texttt{INVOICE}. Note, however, that the clients of the virtual database know only about the virtual tables, and not even the row class knows about the scattering in the real database. There is no static trace of tables \texttt{CUSTOMER}, \texttt{AFFILIATION}, and \texttt{INVOICE}.

Since we are using a corporate customer base to store personal data about all conference attendees, one may ask what to do with participants who have no company affiliation. Such questions are typical for legacy situations where structures are reused for purposes slightly different from the initial intention. In this case we assume that we can invent and store a special \texttt{COMPANY\_CODE} representing private participants and simply leave the corresponding \texttt{COMPANY\_NAME} column blank for these entries.

We will return in the next section to the issue of how queries can be formed in the object model to retrieve persistent object structures. For now, let us just assume that the proper SQL requests will at some point be supplied as an argument to the \texttt{query} feature of the row object.

As already mentioned, it is perfectly legal for a query to return more columns than what is needed to load a given row object. First, there may be a need for the client (in its row action) to check certain data in the cursor structure that will not be transferred to the row object. Second, if a suitable naming convention is used so that the destination of each attribute can be inferred dynamically from its name, a client can use the result of a single SQL query to fill several row objects, one at a time, without the need for unique attribute names across the row classes.

Since non-matching attributes are ignored in each transfer, no conflicts will occur. A standard solution would be to use the SQL renaming facility (SELECT \ldots \texttt{AS}) to give each resulting column a name related to the proper row object attribute, regardless of what names are used in the database tables.

For example, if the data for both a \texttt{REGISTRATION\_ROW} and a \texttt{PERSON\_ROW} were to be returned by the same SQL query, we could use names “\texttt{REGISTRATION\_ROW\$ENTRY\_DATE}” etc. for columns whose destination is the former object and “\texttt{PERSON\_ROWS\$FIRST\_NAME}” etc. for those aimed for the second object.

If the object to be loaded is of type \texttt{NAME} and has an attribute \texttt{attr} the conversion routines would look for columns named “\texttt{NAME\$ATTR}” and transfer the corresponding value. (A SQL statement of type “SQL \texttt{NAMES \texttt{ARE ...$...}}” introducing some character not used for other purposes would ensure unambiguous interpretation.)
11.5 QUERYING A PERSISTENT OBJECT MODEL

Besides a basic mechanism for the retrieval and storage of persistent objects, we need a way to express what objects we want. The general issues of object-oriented query languages are still at the research stage with different directions favoring procedural or declarative approaches. Most concrete proposals from the latter school, so far, have been based on relational algebra (Object SQL); see for example [Kim 1990, Loomis 1991].

However, many commercial applications (probably the majority) do not need the full power of relational algebra to fulfill their functionality, since the types of retrieval performed are pretty much known in advance. Therefore, simplicity and flexibility is often more important than complete generality.

In this section, we will look at a simple approach that can be incorporated with our relational database encapsulation to express a fairly broad class of queries in a very natural way. It can also be used as a basis for automatic translation into SQL statements, provided that the queries are not too complex.

**Query frames**

The idea is to transpose the technique of Query-by-Example [Zloof 1977] to the object-oriented world. Rather than passing a query as a string expressed in some query language, we may simply supply a template describing the retrieval criteria for each attribute of a persistent object. The storage manager responsible for retrieving the corresponding type of object may then inspect the template and return the objects matching the criteria.

A possible scheme would be the following: the client creates a new object of the required persistent type, fills in the attributes that will serve as retrieval criteria, and calls a retrieve operation on the object. The supplier side will then fill in the missing attributes by returning all matching objects, one by one, using the iteration facilities described earlier.

However, there are some disadvantages with this approach. First, basic attributes that are not of reference type (like `INTEGER` or `REAL`) always have values. Therefore, there is no obvious way to signal whether an attribute of this type has been set or not.

If a query result contains a real attribute `temperature`, a value 0 in the template could mean either null (all objects wanted), or zero (only objects of temperature zero wanted). This can be circumvented by defining special values (usually the largest representable negative numbers) and letting clients use these to signify null values for reals and integers.

However, a more severe drawback is that the selection criteria are limited to exact equality. If this is all we need, the approach is nice and simple, but more expressiveness is usually required. So we are going to use a more general
approach, which is to define for each persistent class a corresponding *query frame*.

The query frame of a class is a class containing attributes with the same names, but where all basic types have been replaced by *ANY* (a predefined type to which all other types conform), and all class types have been replaced by the type of its corresponding query frame. The query frame class corresponding to class `REGISTRATION` is shown in figure 11.16.

Using objects of this kind to set up a query frame structure, rather than the objects themselves, opens up new possibilities for expressing criteria.

Retrieval by example

For each basic attribute of the query frame, there are two choices:

1. The frame attribute is set to a value of the same type as that of the corresponding attribute in the model object, in which case the selection criterion becomes exact equality on this value. This is an important option, since we may want to compute the corresponding value dynamically without being forced to convert the result into a string.

2. The frame attribute is set to a string, in which case the criterion may be an expression in any language chosen. For attributes in the model object of string type, we are then faced with a small ambiguity: string values will always be interpreted as criteria expressions rather than as literal values. So if the expression "> 'Johnson'" normally means “all values sorted after 'Johnson'”, some escape conventions are needed to express a literal match of the same string. However, this is not much of a problem, since even very simple string matching languages will need facilities for resolving such situations anyway.
The two types of attribute initialization may be freely mixed in a query. As an illustration, consider the query frames set up in figure 11.17 for the selection of a set of registrations. The query asks for all registrations entered after March 15, 1994, where an invoice was sent, less than $500 has been paid, and the attendee lives in the USA.

![Figure 11.17  Selection of registration objects using query frames](image)

It is also possible to allow lists of query frame structures, each representing a possible selection on the attribute values of the object and its supplier objects. Such a list would then represent logical or of the selection criteria set up by each frame structure.

**Retrieval by key**

It is important for a client to be able to cut off the retrieval of deep structures, so that not everything needs to be transferred at once. Particularly, there may be recursive object structures that simply cannot be retrieved in just one SQL statement. To this end, we employ the convention that whenever an attribute of class type (representing a reference to another object) is set to Void in a query frame, the corresponding object is not retrieved. This is the case for selected_tutorials in figure 11.17. If the attribute had been initialized with an empty SET [TUTORIAL], the tutorial objects would have been retrieved too for each registration.

When an object reference is cut off in a query frame by initializing an attribute of class type to Void in a retrieval by example, it does not necessarily mean that the client will not be interested in the corresponding object. It may be wanted after some inspection of the retrieved data.
Assume a terminal operator scans through a large number of registration objects without retrieving the corresponding `attendee` fields until a particular registration is reached, at which point the personal data suddenly becomes interesting. It would then be awkward if the application had to have a provision for reinitializing a parallel `REGISTRATION_FRAME` object (this time with the `attendee` reference initialized to an empty `PERSON` object rather than `Void`), and then retrieve the same registration once more in order to get the personal data.

However, this will not be needed, since even if the `PERSON` object was not retrieved the first time, the `REGISTRATION_MANAGER` has the key information in the corresponding `REGISTRATION_ROW` object to get it directly from the database.

### 11.6 PERSISTENT OBJECT MANAGEMENT

We will now take our design one step further, and establish enough detail to outline a full scenario from start to end of a simple persistent object retrieval. To this end, we extend the upper level of the preliminary sketch in figure 11.14 and introduce a few more classes as shown in figure 11.18 to capture the general principles involved.

![Figure 11.18 Persistent object management](image)

Our aim is to keep as much persistency detail as possible out of the class definitions of the model objects. Therefore, the only static differences between a class whose objects are potentially persistent and one whose objects are just transient are the following two.
First, a persistent class must inherit from the class \textit{PERSISTENT} (see figure 11.18). This will enable clients to invoke retrieval operations on the objects and iterate through sets of matching instances. Second, it will need to redefine the signature of \textit{retrieve_by_example}. The argument supplied as retrieval criteria for a persistent \textit{REGISTRATION} object, for example, must be defined as \textit{REGISTRATION\_FRAME}.

Each type of persistent object is retrieved and stored in the underlying database by a corresponding manager class, and all manager classes inherit from \textit{STORAGE\_MANAGER}. The idea is not to build static knowledge into the persistent classes by specifying the exact type of manager needed to take care of the corresponding objects. Instead, there will be a dynamic mapping available, so that persistent objects can invoke their proper manager by simply stating their own type. Since the class name is already a unique type identification, a mapping from class name strings to the corresponding manager will be enough (to keep the discussion simple, we assume that the persistent classes are non-generic).

The class name of an object can often be obtained dynamically from predefined system classes in many object-oriented environments. One of two standard techniques may then often be used for manager routing:

- If there are facilities in the environment to create a new instance of a class directly from the class name, we only need a mapping to the class name of the manager.

- If this is not possible but there is an “object cloning” facility available, we may instead use object templates. At system initialization, one instance of each persistent manager class is created to serve as a cloning template, and a table of type \textit{TABLE [STORAGE\_MANAGER, STRING]} is set up to map each persistent class name into a reference to one of the template objects. The returned reference is then forwarded to a cloning facility, which will instantiate a new copy of the object.

The class \textit{MANAGER\_TABLE} in figure 11.18 is assumed to take care of the mapping, using some suitable technique. When called upon to access persistent data, the features of \textit{PERSISTENT} will thus look up the proper manager and establish the bidirectional client link between the object and its manager.

It is important to note that although the two classes \textit{PERSISTENT} and \textit{STORAGE\_MANAGER} depend on each other, they are independent of which subtype the other party will have. The specific manager class that will do the actual conversion work must of course have full access to the attributes of the persistent object, so \textit{REGISTRATION\_MANAGER} will statically depend on \textit{REGISTRATION}, but not the reverse (see figure 11.18).
Also, in this design we have assumed one manager for each persistent object type. However, this is not necessary when dynamic routing is used. If there are a large number of persistent classes in a system, their management will probably tend to repeat typical patterns, and it may then be desirable to have fewer, more general, managers to take care of groups of persistent object types.

We now proceed to look at the collaborating features of the two common ancestors of persistent classes and storage managers respectively.

**Persistent objects**

The interface of class *PERSISTENT* is shown in figure 11.19. The first time a persistency operation is called on a persistent object, the appropriate manager template will be located through a routing table shared by all persistent objects. A new storage manager will then be created and attached to the *manager* attribute of the persistent model object, and a back reference assigned in the manager object.

Three forms of retrieval, *retrieve_by_example*, *retrieve_by_command*, and *load_from_cursor*, are available for persistent objects. All three commands will be transparently forwarded to the appropriate storage manager without any processing. Note that the only thing that needs to be changed when the feature *retrieve_by_example* is redefined in a persistent class is the type of the query frame argument. All implementation logic will reside in the corresponding manager.

The first retrieval form implements the high-level selection criteria suitable for application clients, which should be independent of any lower-level access details. However, even the storage managers should know the low-level details only of the objects they manage. Note that this includes what is defined by the corresponding persistent class, but does *not* include what is defined by any of its supplier classes.

For example, to retrieve a *REGISTRATION* object, the registration manager will (in most cases) need to retrieve a corresponding *PERSON* object referred to by *attendee*. However, it would be most unfortunate if the mapping of the attributes of class *PERSON* into attributes of the virtual relational database (or even worse, to the real database) had to be statically known by class *REGISTRATION_MANAGER*.

If this were the case, we would need to create and maintain manager implementations not only for each persistent class, but also for each combination of a persistent class using another one as client. In a system with a large number of persistent classes, the situation would soon become unmanageable.

One improvement would be to let the registration manager call a *PERSON_MANAGER* to have the *attendee* part retrieved and translated.
However, this would create a lot of static dependencies between different managers (somehow mirroring the dependencies between the corresponding model objects, but with enough differences to create maintenance problems). So a better solution is to go a step further, and always channel any persistency operation through the persistent objects themselves.

This is where the alternative retrieval forms `retrieve_by_command` and `load_from_cursor` come in. They are both meant for storage managers rather
than application clients. The arguments supplied when invoking these operations can be used for communication between managers, but the routing will be done by the corresponding model object, so that no unwanted static dependencies are created.

The iteration features are similar to the ones already discussed for the lower-level clusters. An application can attach an action object (usually itself) and then receive a callback for each retrieved object instance matching the selection criteria.

Storage managers

A storage manager translates persistent data between a model object and a corresponding virtual relation (in case the instances are stored in a relational database, as for `REGISTRATION` and `PERSON` in our example) or some other storage (in case the instances are stored elsewhere, as for `TUTORIAL`). We will only discuss the relational aspect in this case study.

The three forms of retrieval are different. The first, `retrieve_by_example`, will cause the manager to read the supplied query frame object (or object structure, if “inner” frame objects are also included) and use the attribute information to find a suitable SQL query that will return the data required to set up the matching objects.

As was argued earlier, it is desirable to minimize the static dependencies on the exact organization of the real database, which is why we introduced a virtual relational database represented by the row classes. However, the SQL statements certainly need to be phrased in terms of the current database schema, so how can we avoid becoming statically dependent on that schema when putting the queries together?

We will return to the issue of automatic generation of SQL queries in the concluding section, but for now we will only assume that whatever steering information needed to dynamically construct the SQL statements that may occur in our system (not always that many different types) is somehow maintained outside the compiled executables. Applications will thus not need recompilation when schema changes occur that do not affect the logical organization of the persistent objects, which is our goal.

Unless we come up with a good automatic translation for a broad class of queries, the stored tables may have to be structured ad hoc and perhaps not be so trivial to maintain. However, even with a low level of automatic support, we should be better off than if we are forced to change our compiled classes for each minor schema change.

We assume that a class `SQL_MAPPINGS` will encapsulate a set of mapping primitives, which will be used by the managers to dynamically build the required
SQL statements. In an ambitious approach, the mapping data required would probably be stored with the objects in the relational database.

The retrieve_by_command feature is mainly used when a manager needs to obtain an object through its primary key. As was explained in an earlier section, a REGISTRATION_FRAME query object may be set up to cut off the retrieval of personal data by initializing the attendee reference to Void. The registration manager will then only retrieve the basic attributes of REGISTRATION, but will keep the primary key to the corresponding PERSON object in case it is requested later.

A common scenario may be a terminal operator quickly browsing through a large number of registrations with retrieval of the personal data turned off. When certain field values appear on the screen, the operator becomes interested and orders the personal data to be filled in. The registration manager will then typically be in the middle of its callback execute routine processing the current instance of an iterate_on_result, and have a truncated REGISTRATION as model_object.

The application object (which receives the order while waiting for input in its callback action) then creates an empty PERSON_FRAME object, assigns it to the attendee attribute of the REGISTRATION_FRAME object, and issues a new retrieve_by_example on the same registration object.

The registration manager then detects that a new retrieve has been issued in the middle of an iteration, which leads to a different action. Rather than as a request for a new retrieval of model objects, the query is now understood as a request to retrieve more of the deep structure of the REGISTRATION instance already available. The degree of depth in such a new retrieval is again controlled by the values (void or non-void) of the non-basic attributes of the frame object and its suppliers, recursively. In the case of PERSON, there is no further structure to retrieve.

So the manager rescans the query frame and detects that the attendee attribute is no longer void and should be filled in. The registration manager then generates a suitable SQL query to obtain the missing data and invokes a retrieve_by_command on an empty PERSON object with the SQL string as argument. It then attaches itself as action object and starts a separate iterate_on_result, which will only return one PERSON object since it was retrieved by its primary key. The translated person object is then assigned to the registration object and control is returned to the application execute routine, which displays the missing data on the screen and the operator can continue browsing.

(The conventions just described represent of course a rather special design decision, but the idea is to convince the reader that reasonably clean solutions are indeed possible.)
Finally, the third form of retrieve, *load_from_cursor*, directs the manager to load its *model_object* with relational data obtained by another manager.

When composite objects like *REGISTRATION* are to be retrieved from the database, the most efficient way is usually to let the database server do the required joins to obtain both the registrational and personal data at once. This means that a registration manager will construct the full query and supply it to a *REGISTRATION_ROW* object, which will call the database server and get a set of table rows in return.

The registration manager then invokes a *load_from_cursor* on the row object to load all basic attributes of the registration, and the manager can translate these into the registration object. However, the person attributes for the *attendee* supplier object remain, and they cannot be loaded and translated by this manager, since we do not want cross-dependencies on internal formats.

Instead, the registration manager simply creates an empty *PERSON* object and invokes *load_from_cursor* on it, supplying itself as cursor holder. The person object does not know anything about cursors, but it can propagate the request to a *PERSON_MANAGER*, which will then access the supplied manager argument, extract the corresponding *REGISTRATION_ROW*, and supply it to a *PERSON_ROW* as argument of an *attach_to_query*.

The person manager invokes *load_from_cursor* on the *PERSON_ROW* object, which (since it has been reattached) will then transfer data from the cursor held by the registration row, and then translates the resulting row attributes into the person object. The registration manager can now use the retrieved person object to complete its registration object.

**A full retrieval scenario**

We are now ready to present a complete scenario describing how a persistent registration object is retrieved from the database. A dynamic object diagram with its accompanying scenario box can be found in figure 11.20. However, for the interested reader, we will also go through each step in more detail below and mention the operations involved.

A typical scenario would proceed as follows.

1. A client initializes a *REGISTRATION_FRAME* with attribute *attendee* attached to a *PERSON_FRAME* object to signify that the personal data of each registration should also be retrieved. The attributes (of both frame objects) whose values are to be part of the selection criteria are set to mirror the conditions. The client then invokes *retrieve_by_example* on a *REGISTRATION* object, supplying the query frame as argument.

   The registration object invokes a corresponding *retrieve_by_example* on its *REGISTRATION_MANAGER* passing the query frame as argument.
Scenario: Retrieval of persistent registration objects

1–2 The client initializes a query frame with selection criteria, and asks a registration object to retrieve matching instances.

3–7 The registration object calls a registration manager, which inspects the query frame and sends the appropriate SQL statements to a registration row object. The row object forwards the query to the database server, which returns a cursor structure.

8–10 The client tells the registration object to iterate on the result, that is load the matching objects one by one, each time giving the client a callback to process the instance. The request is forwarded to the registration manager, which starts an iteration on the registration row, supplying itself as action object.

11–12 The registration row starts an iteration on the cursor structure, forwarding the manager as action object. The manager action is called with the cursor pointing to the data of the first matching object.

13–19 The registration manager tells the registration row to load itself from the cursor, and sets the basic attributes of the registration object by translating from the corresponding row attributes. It then creates a person object and tells it to load itself from an existing cursor, supplying itself as cursor holder.

20–23 The person object forwards the load request to a person manager, which obtains the registration row from the registration manager argument. The person manager tells a person row to attach itself to an existing cursor structure held by the registration row.

24–31 The person manager tells the person row to load itself from the current cursor and sets the attributes of the person object. The registration manager completes the registration by inserting a reference to the person object, and calls the client action routine for application processing.

Figure 11.20 Typical retrieval scenario
registration object has no manager, a new one is created and attached using
the shared routing table.

2. The registration manager then translates the query frame attribute values to
appropriate SQL statements, and calls `query` on a `REGISTRATION_ROW`
with the query string as argument (see figure 11.15 for the interface of row
classes). The `REGISTRATION_ROW` attaches itself to a `DB_QUERY`
object and calls its `query` operation passing the SQL string. A set of table
rows is then returned from the database server (see figure 11.8 for the
interface of the database encapsulation).

3. The client uses `set_action` on a `REGISTRATION` to attach an action object
for processing (usually itself) and then calls `iterate_on_result` on the
registration, which is passed to the `iterate_on_result` of the manager. The
registration manager calls `set_action` on the `REGISTRATION_ROW`
supplying itself as action object, followed by an `iterate_on_result` on the
row.

4. The `REGISTRATION_ROW` transfers the manager as action object to the
`DB_QUERY` and calls its `iterate_on_result`. The `DB_QUERY` creates a
`DB_RESULT` representing the first matching table row and invokes the
`execute` callback in the registration manager. The manager then calls
`load_from_cursor` on the `REGISTRATION_ROW`, which then calls
`load_object` through the `cursor` feature of `DB_QUERY`, supplying itself as
receiving object.

5. The `DB_RESULT` object loads matching attributes of the first row of the
result into the `REGISTRATION_ROW` object. The `execute` routine of the
registration manager then proceeds to translate the row object attributes
into the `REGISTRATION` object.

6. All basic attributes of the registration have now been loaded, and the
registration manager proceeds to retrieve person data while still
performing its `execute` callback routine. The registration manager creates
a new `PERSON` object and invokes `load_from_cursor` on the empty object.
The cursor holder passed as argument to `load_from_cursor` is a reference
to the `REGISTRATION_MANAGER` itself.

   The `PERSON` object then invokes `load_from_cursor` on its
   `PERSON_MANAGER` (attached to the person object via feature `manager`).
The `REGISTRATION_MANAGER` reference just received by the `PERSON`
object is again passed as cursor holder in this second call.

7. The `load_from_cursor` command in the `PERSON_MANAGER` starts by
getting a reference to a `REGISTRATION_ROW` through the `row_object`
feature of the registration manager passed in as argument. It then invokes attach_to_query on the PERSON_ROW referred to by feature row_object, passing as argument the registration row. The effect is to connect the person row to the cursor structure held by the registration row.

When this has been done, the person manager issues a load_from_cursor on the PERSON_ROW. The person row calls load_object on the cursor of its newly connected DB_QUERY and supplies itself as receiver argument. The DB_RESULT object now loads matching attributes of the first row of the result into the PERSON_ROW object.

8. The person manager translates the person row object attributes into the PERSON object. Control is then returned to the execute routine of the registration manager, which finishes its work by attaching the person object to the registration, which then becomes complete. It then invokes the callback routine of the client.

9. The execute of the client then processes the returned persistent object. When the routine exits, control is returned to the execute of the registration manager, which returns it to the DB_QUERY, which moves the cursor to the next resulting table row and again invokes a callback in the manager.

10. The above scenario continues until no more resulting rows remain, or the application client has signaled termination by setting the over feature of its action object to true. When this happens, the manager will propagate the decision by setting its own over feature, and this will make the iterate_on_result of DB_QUERY return control instead of issuing a callback. The corresponding iterate_on_result of the manager also returns, and the client may proceed at the point of its initial iteration call to the registration object.

We will conclude the case study with a discussion on how much automatic support is feasible for the generation of SQL queries.

11.7 AUTOMATIC GENERATION OF SQL STATEMENTS

Since the SQL queries must be expressed in terms of the real database schema, we do not want to have them hard-wired into our software applications. So we are faced with two problems:

- We need to obtain dynamically the mapping between our virtual database and the real database in order to generate a SQL selection statement that will return the proper columns for our row classes.
• We need to find a way for application clients to express selection criteria solely in terms of model object attributes, which can be dynamically translated into SQL statements on the real database.

These are challenging requirements which are not easy to meet. If we were to invent an object-oriented query language completely unrelated to SQL, the dynamic translation would most likely become intractable. However, if we impose some suitable restrictions on the permitted queries and on the relational mappings used, new possibilities open up.

Model queries

The idea is to use SQL syntax transposed to the model object attributes. If the strings assigned to the attributes of a query frame object are valid SQL expressions restricting the corresponding model object attributes, a simple translation scheme is possible. With this approach, the query frames of figure 11.17 are equivalent to the query shown in figure 11.21, expressed in an object-oriented SQL notation.

```
select
    registered_at, amount_paid, invoice_sent, confirmed,
    paper_sessions, attendee.name, attendee.affiliation,
    attendee.address, attendee.postal_mail, attendee.email,
    attendee.phone, attendee.fax
where
    registered_at > '1994-03-15' and
    amount_paid < 500 and
    invoice_sent = true and
    attendee.address LIKE '%USA%'
```

Figure 11.21 An object-oriented SQL query

Such a notation may also be used by application clients as an alternative to the query frames of retrieve_by_example. A selection is then passed as a string to retrieve_by_command. If all attributes that are part of the selection criteria in the model object are representable as SQL expressions built from columns in the real database, the translation is reduced to simple substitution.

We may still allow some criteria involving object attributes that cannot be transformed to SQL expressions using the database schema, but then these criteria cannot be part of the selection sent to the database server. A correspondingly larger number of table rows will thus be retrieved, and the ones not matching the non-translatable criteria filtered out a posteriori by the manager of the persistent object type.
Attribute mappings

The mapping facilities that are needed for storage managers to make automatic conversions are encapsulated in the class \textit{SQL\_MAPPINGS}, whose interface is outlined in figure 11.22. All mappings are between strings and can be maintained as simple editable tables and, for example, be stored as persistent strings in the database.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textit{SQL\_MAPPINGS} & \\
\hline
\texttt{db\_select}: STRING & -- Table columns of real database \\
 & -- corresponding to persistent\_class \\
 & $\rightarrow$ persistent\_class: STRING \\
\hline
\texttt{db\_from}: STRING & -- Tables corresponding to persistent\_class \\
 & $\rightarrow$ persistent\_class: STRING \\
\hline
\texttt{db\_join\_condition}: STRING & -- Table join condition in real database \\
 & -- corresponding to a non-basic attribute \\
 & -- of a persistent class. Argument format: \\
 & -- "CLASS\_NAME.attribute\_name" \\
 & $\rightarrow$ attribute: STRING \\
\hline
\texttt{db\_attribute\_expr}: STRING & -- Table column expression in real database \\
 & -- corresponding to a non-basic attribute \\
 & -- of a persistent class. Argument format: \\
 & -- "CLASS\_NAME.attribute\_name" \\
 & $\rightarrow$ attribute: STRING \\
\hline
\texttt{db\_update\_pattern}: STRING & -- SQL template for generation of real database \\
 & -- update from attributes of row\_class \\
 & $\rightarrow$ row\_class: STRING \\
\hline
\end{tabular}
\end{center}

\textbf{Figure 11.22} Query generation primitives

The first four features return information needed to build queries that can be sent to the database server. A generated query consists of three parts:

- The table columns that need to be selected to provide enough information for building each persistent instance.
- The join conditions for the natural joins to be performed between tables representing a client and tables representing a supplier. For example, class \textit{REGISTRATION} needs to join its tables with those of class \textit{PERSON} to obtain the data corresponding to the attribute \textit{attendee}. 
The selection criteria expressed as SQL conditions on the retrieved database table columns.

The target columns of the selection are precisely the attributes of the corresponding row classes. The call `db_select("REGISTRATION")` would return:

```
"REGISTRATION.PERSON_CODE as REGISTRATION_ROW$PERSON_CODE,
REGISTRATION.ENTRY_DATE as REGISTRATION_ROW$ENTRY_DATE,
REGISTRATION.PAPER_SESSIONS as REGISTRATION_ROW$CONFERENCE,
REGISTRATION.TUTORIAL1 as REGISTRATION_ROW$TUTORIAL1,
REGISTRATION.TUTORIAL2 as REGISTRATION_ROW$TUTORIAL2,
REGISTRATION.TUTORIAL3 as REGISTRATION_ROW$TUTORIAL3,
REGISTRATION.TUTORIAL4 as REGISTRATION_ROW$TUTORIAL4,
REGISTRATION.CONFIRMATION_DATE as
  REGISTRATION_ROW$CONFIRMATION_DATE,
REGISTRATION.INVOICE as REGISTRATION_ROW$INVOICE,
INVOICE.ISSUED_DATE as REGISTRATION_ROW$ISSUED_DATE,
INVOICE.AMOUNT_PAID as REGISTRATION_ROW$AMOUNT_PAID"
```

If the selection includes personal data, a `db_select("PERSON")` will also be needed to add the corresponding columns to the format of the retrieved rows.

Besides the selected columns, we need to accumulate all tables that participate in the resulting selection. These could in principle be extracted from the former strings, since we store all column selections qualified by table name to avoid any name clashes. However, sometimes table alias names must be used for unambiguous reference (see the next section).

We therefore store the table reference strings separately, and the calls `db_from("REGISTRATION")` and `db_from("PERSON")` yield, respectively:

```
"REGISTRATION, INVOICE"
"CUSTOMER, AFFILIATION"
```

Next, we need to build the restriction clause on the rows initially selected. This must include the corresponding join condition for each “inner” object. Calling `db_join_condition("REGISTRATION.attendee")` would return:

```
"REGISTRATION.PERSON_CODE = CUSTOMER.CUSTOMER_CODE"
```

In fact, when the basic attributes of a class are stored in more than one table, there will be join conditions involved besides the ones introduced by “inner” objects. Therefore, entries may also have to be stored for the class names themselves, not just for each of their non-basic attributes. This is the case for both our classes and the calls `db_join_condition("REGISTRATION")` and `db_join_condition("PERSON")` will return the following strings:

```
"REGISTRATION.INVOICE = INVOICE.INVOICE_CODE"
```
"CUSTOMER.COMPANY = AFFILIATION.COMPANY_CODE"

which must also be part of the restriction clause. Finally, before completing the
generated query by appending the selection criteria supplied by the client object,
each model object attribute occurring in these criteria needs to be substituted by
a corresponding SQL expression in terms of the real database columns. To this
end, the calls:

```sql
    db_attribute_expr("REGISTRATION.registered_at")
    db_attribute_expr("REGISTRATION.amount_paid")
    db_attribute_expr("REGISTRATION.invoice_sent")
    db_attribute_expr("PERSON.address")
```

will return the following strings, respectively:

```
"REGISTRATION.ENTRY_DATE"
"INVOICE.AMOUNT_PAID"
"INVOICE.ISSUED_DATE is not null"
"PERSON.STREET || ', ' PERSON.ZIP_CODE || ', ' PERSON.CITY || ', ' COUNTRY_NAME"
```

**Self-joins**

Assume a conference system which also keeps track of alternative arrangements
(often called spouse programs) for people accompanying attendees. A possible
model is shown in figure 11.23.

![Diagram](image)

Figure 11.23  Simplified model for **ATTENDEE** and **COMPANION**

An ancestor class **PERSON** contains common attributes, while **ATTENDEE**
and **COMPANION** add the needed extensions.

If for some legacy reasons we want to use only one relational table to store
both types of object, letting each row represent either a conference attendee or
someone accompanying an attendee, such a table containing personal data could
look like the one in figure 11.24. A row representing an attendee would leave
the selected tour column blank, while a companion row would have companion
code zero.
At first glance, it may seem as though an ATTENDEE object would need to be built from two SQL queries: the first selecting the attendee row and the second selecting a possible companion based on the retrieved value of the companion code column. However, this is not necessary.

We could enter a companion row with blank name and address fields, and let it represent the “empty” companion. In each row representing an unaccompanied attendee, the corresponding key value would be used as companion code, thus ensuring referential integrity.

The SQL selection below could then be generated (column renaming omitted for brevity), and the resulting rows would contain all data needed to regenerate the complete ATTENDEE objects, one by one.

```sql
SELECT ATTENDEE.NAME, ATTENDEE.ADDRESS,
      COMPANION.NAME, COMPANION.ADDRESS, COMPANION.SELECTED_TOUR
      REGISTRATION.REGISTERED_AT, ...
FROM PARTICIPANT AS ATTENDEE, PARTICIPANT AS COMPANION,
     REGISTRATION
WHERE ATTENDEE.name = 'John Hopkins' AND
      ATTENDEE.COMPANION_CODE = COMPANION.PARTICIPANT_CODE AND
      ATTENDEE.REGISTRATION_CODE = REGISTRATION.REGISTRATION_CODE
```

**The generation scheme**

To summarize, the generation of a SQL query using the features of SQL_MAPPINGS would proceed as follows:

1. Scan the query frames to find out how many objects must be instantiated from each table row returned by the SQL query. This is recursively controlled by the client, which can either initialize a reference to a non-basic object in the query frame, or leave it as Void.
2. Construct the **select**-part by appending the database attribute list of the corresponding row class for each object that is to be loaded from a resulting table row. This is done through repeated calls to `db_select`. (Here, as in the following steps, we implicitly assume that any needed keywords and comma separators are also inserted.)

3. Construct the **from**-part by appending the database tables involved for each object (repeated calls to `db_from`).

4. Start constructing the **where**-part by appending the join condition for each model object, as well as for each attribute referring to a participating “inner” object (like `attendee` in our example). This is done through repeated calls to `db_join_condition`.

5. Translate all selection criteria supplied by the client query frames that are expressed in terms of model object attributes into equivalent criteria understandable by the database server. This is accomplished by substituting the corresponding SQL expressions of the real table columns using calls to `db_attribute_expr`.

6. Complete the **where**-part by appending the resulting criteria strings.

### Updating the database

The generation of SQL statements for updates and insertions is much simpler. Since all data, including primary keys, is available in the row objects, we only need to store SQL templates to be filled in with current attribute values. The call `db_update_pattern("REGISTRATION")` would return:

```sql
update REGISTRATION set 
"ENTRY_DATE = :entry_date, CONFERENCE = :conference, 
TUTORIAL1 = :tutorial1, TUTORIAL2 = :tutorial2, 
TUTORIAL3 = :tutorial3, TUTORIAL4 = :tutorial4, 
CONFIRMATION_DATE = :confirmation_date, INVOICE = :invoice 
where PERSON_CODE = :person_code"
```

Preceding a name by a colon is the usual SQL placeholder notation to signify that a value will be dynamically substituted. The manager could either use the pattern to fill in a complete SQL statement for immediate execution, or put it in a stored procedure to be furnished by attribute values using the dynamic SQL conventions for argument passing.

The `issued_date` and `amount_paid` attributes are not part of the pattern, since in our legacy example we assume that these values will only be read by the conference system, while the corresponding updates are done by a separate invoicing system.
Moreover, we have omitted insert and delete from our design for simplicity, but the mappings for these features can be handled similarly.

11.8 FULL STATIC ARCHITECTURE

Finally, we present the complete architecture of our persistency layers in figure 11.25, showing how it all fits together. As can be seen, the different layers are well separated and the static dependencies between them have been reduced to a minimum.

![Diagram of Persistent object mapping: full static architecture]

Figure 11.25  Persistent object mapping: full static architecture