5. Design

The persistence framework design is presented in Figure 5.1. The design classes are distributed into four subsystems. **Relational Data** handles relational data in the form of tables and records. **Persistence Mechanism** contains classes that encapsulate access to the underlying persistence mechanism. **Broker** provides object persistence services and is the main framework subsystem. Not all the design classes are visible – the user only sees classes in the **Framework** facade, which forms the user interface for the persistence framework.

![Design model of the persistence framework](image)

*Figure 5.1 – Design model of the persistence framework*
5.1 OIDObject and DataObject

Shown in Figure 5.2, these two interfaces, OIDObject and DataObject provide a standard method of accessing objects that have name or data associated with it. *getOID()* returns the name or identifier associated with the object, while *setOID()* provides a way to set or change the name. As for DataObject, the *getData()* method returns the data object associated with it. To set or change the data object, *setData()* is used.

![OIDObject and DataObject interfaces](image)

Figure 5.2 – Base objects class diagram

5.2 Relational Data: TableObj, Record, and Column

The classes TableObj, Record and Column, form the data structures that are native to a relational database. Shown in Figure 5.3, a TableObj consists of Records, which in turn, consists of Columns. Both TableObj and Record are container classes that implement the OIDObject interface which represent the table name and record name respectively. These classes provide methods to add, remove, and retrieve items contained within it. They may be empty.

For the TableObj class, *add()* adds Records and returns a boolean value to indicate whether the process was successful or not. *get()* retrieves records by name or by index. *remove()* deletes them, returning either the removed record or a boolean success value. The *select()* method is used for simple queries of the table. The first parameter is *colname* which specifies the name of the column, while *cond* is the object to be used in the comparison. The method returns the first record for which *colname=cond*. To clear the table of all records, *clear()* is used. *size()* is called to return the number of records in the
table. A TableObj is constructed by just passing the table name to the constructor.

The Record class behaves similarly, with methods add(), get(), remove(), clear(), and size(), except that Columns are involved rather than Records. A convenience method add(colname: String, data: Object) is used to directly construct and add a Column to the record.

The Column class implements the OIDObject and DataObject interfaces, which represent the column name and column data respectively. The column data object is the actual value contained in the column itself, and should be in the form of a simple Java data

---

![Class diagram](image)

**Figure 5.3 – Relational Data class diagram. (Attribute and parameter types are Java classes.)**
type, such as an integer, string, floating point number, etc. The constructor accepts the column name and data object as parameters. A default constructor is also provided.

5.3 Persistence Mechanism: FlatFile, Email, JDBC, and ObjectStore

The PersistenceMechanism interface shown in Figure 5.4 is implemented by any persistence mechanism class that wishes to communicate with the PersistenceBroker. With this standard interface, the broker can support virtually any form of persistence.

![Persistence Mechanism Diagram]

**Figure 5.4** Persistence Mechanism class diagram (Attribute and parameter types are Java classes)
mechanism transparently. The methods contained perform tasks such as connecting, querying, saving, restoring and deleting tables.

A persistence mechanism class each implements a standard constructor taking three arguments: the data source name, username, and password. The data source name can be a filename, database name, or email server, depending on the type of mechanism. The username and password are used to authenticate a user with the mechanism, if required.

To obtain information about the persistence mechanism's capabilities, a few methods can be used. The \texttt{getType()} method returns a string identifier for the mechanism type, and is typically a short word such as "file", "jdbc", and so on. \texttt{supportsTransactions()} returns a boolean value that indicates whether the mechanism supports transactions, while \texttt{supportsSQL()} returns a boolean to indicate support for queries.

The \texttt{open()} method is used to connect to the data source name, whereas \texttt{close()} is used to disconnect. Both return integer values to indicate the success of the operation; zero indicates successful completion, while non-zero values indicate failure.

The table methods allow for creating, reading, updating, and deleting tables from the mechanism. All of them use the \texttt{TableObj} class for accepting input and returning output data. \texttt{createTable()} takes a table as a parameter and creates a new table based on the data passed to it. \texttt{readTable()} retrieves a table from the persistence mechanism. \texttt{updateTable()} updates an existing table with new data, and \texttt{deleteTable()} removes an existing table with the table name specified as a parameter. The create, update, and delete methods return an integer, zero indicating success, non-zero otherwise. The read method returns data in a \texttt{TableObj} object.

To manipulate individual records rather than whole tables, \texttt{addRecord()}, \texttt{getRecord()}, \texttt{updateRecord()}, and \texttt{deleteRecord()} can be used. These methods are similar to the table methods except that now the parameters passed must include the column name, comparison object, and record object.

The \texttt{update()} method is used to update the value contained in a certain field of a record in a table. The table name, column name to be updated, the new value, the
comparison column, and the comparison object have to be specified. It then searches for
the first record satisfying the comparison column and object and sets the update column to
the new value.

If the mechanism supports querying, as indicated by the supportsSQL() method,
then the query() method can be called with a SQL statement as a parameter. The statement
is passed to the mechanism to be executed, and the results are returned as a TableObj. If
an error occurs, or the mechanism does not support queries, then a null value is returned.

Support for transactions also depend on the mechanism
(supportsTransactions()), and are provided through the startTransaction(),
endTransaction(), and abortTransaction() methods, which start, commit, and rollback
transactions respectively. The transaction isolation level can also be set using
setTransactionIsolation(). This determines the level of locking of the database during the
transaction. getTransactionIsolation() retrieves the transaction isolation level, while
supportsTransactionIsolationLevel() checks whether the database supports the specified
isolation level.

The specialised class FlatFileMechanism enables tables to be stored in a flat file
database, that is, one table per file. This mechanism does not require any database system,
only read/write access to a filesystem.

EmailMechanism encapsulates access to an e-mail server. Tables are stored much
like flat files, but in this case each e-mail represents a single table. It uses the standard
Internet Simple Mail Transfer Protocol (SMTP) for sending mail, and the Post Office
Protocol (POP) for retrieving them. As such, the data source name should contain the
proper POP and/or SMTP server addresses, e.g. siswazah.fsaktm.um.edu.my, or
202.185.108.1.

The JDBCMechanism class allows tables to be stored in relational databases that
are supported under Java Database Connectivity (JDBC). It is a standard interface from
which various databases can be accessed such as Oracle, Microsoft SQL Server, IBM’s
DB2, and other databases for which JDBC drivers are provided. Open Database
Connectivity (ODBC) databases are supported through the JDBC-ODBC bridge driver.
This allows the persistence broker to support most commonly used relational databases.
ObjectStoreMechanism uses Object Design’s ObjectStore object-oriented database to store TableObjs. Although it is possible for the persistence broker to store the original unmapped objects directly into it, it is not desirable because ObjectStore cannot store objects transparently. To store objects, each compiled class will need to be run through a post-processor, and classes that are associated with each other need to be processed together in a batch. This will require user intervention and violates the encapsulation principle and the layered architecture of the persistence framework. To overcome this, the framework will do the required post-processing for the TableObj class, so the user is insulated from this process and the framework can operate transparently.

5.4 Broker: PersistenceBroker and Supporting Classes

The Broker subsystem, shown in Figure 5.5, forms the backbone of the persistence framework and performs the core functions required for persistence. It handles all the use cases described in Chapter 4. It is composed of several helper classes to perform the object-to-relational mappings.

OIDGenerator generates unique and immutable object identifiers for each object that is passed to its generate() method. Each identifier should be unique in the context of the persistence mechanism and no two similar identifiers can be generated. It should also be immutable, in the sense that it does not change over time with repeated executions of the application, so that it can be stored persistently and be retrieved at a later time.

The ObjectCreator class is used by the persistence broker to instantiate objects of a specified class type. The create() method accepts the classname as an argument, and in the case where the class is a Java inner class, the outer containing object is required as well. The object is then constructed and returned. If the object cannot be constructed or an error occurs, then a null object is returned.

The ObjectMapper class implements a one-to-one map between user objects and object identifiers. The mapping is transparent and hidden from the user, thus enabling the user to store any object without explicitly providing identifiers for each object. The mapping is implemented using a weak reference to the user object. The user application
Figure 5.5 – PersistenceBroker class diagram. (Attribute and parameter types are Java classes)
will maintain a strong reference to the object. The object will exist as long as there are
strong references to it, so when it is destroyed within the user application, it will be
removed from the map as well. This is shown in Figure 5.6. The `getID()` method returns
the identifier of the object passed to it, while `getObject()` returns the object associated with
the identifier passed to it. Both methods return null if the corresponding mapping does not
exist. Adding object-identifier mappings are done through the `put()` method, and
removing them through the `remove()` method. They both return the previous identifier for
which the object was mapped, or null if there was no previous mapping.

The PersistenceBroker is the main class that the user application interacts with.
Only one instance of it is needed to provide persistence services for the application. The
application must first call the `connect()` method, passing the data source name, username
and password to establish a connection to a persistence mechanism. The data source name
should be a string of the following format, "[mechanism type] : [data source name]", where the mechanism type is the identifier returned by
PersistenceMechanism::getType(), while the data source name is the actual name of the
data source of the said mechanism type. Some examples of the format of data source

![Diagram](image)

**Figure 5.6 – Object mapping process**
names are "file:c:/tmp/database", "email:pop.mail.com,smtp.mail.com", "jdbc:DataBase". The `connect()` method will first parse the data source name for the mechanism type, determining whether it is in the list of supported mechanisms. If it is supported, it will instantiate a `PersistenceMechanism` of the said type and call its `open()` method. If the connection is successful, an integer handle is returned, otherwise -1 is returned. One `PersistenceMechanism` will be instantiated for each connection. The handle represents the connection and must be used in all further interactions with the `PersistenceBroker`.

To disconnect from the persistence mechanism, the `disconnect()` method is called with the handle as a parameter. The method then calls the `PersistenceMechanism::close()` method and returns zero on success, non-zero on failure.

5.4.1 Saving

The `save()` method takes the connection handle, object, and an identifier as its parameters. The identifier is an arbitrary string assigned by the user to serve as the object’s identifier. Thus the object is a named persistent root object. The method then calls the `deepSave()` method, which obtains all objects that are reachable by this persistent root and maps them to individual records. The records are then saved into their respective tables by calling `PersistenceMechanism::updateTable()`. The activity diagram is shown in Figure 5.7. The whole save process effectively decomposes every reachable object in the graph into primitive types such as integers, characters, strings, etc., and saves those primitive types into columns of a table in the persistence mechanism. Relationships between objects are saved as foreign keys as described in earlier chapters. Three maps are used during saving, `objmap` is the object-identifier map described earlier, `refmap` keeps track of the reference count of objects and also serves as a marker to prevent traversal loops as mentioned in Section 4.2.2.2, and finally `savemap` stores the mapped object records to be saved. Object identifiers are generated by `OIDGenerator::generate()` automatically for new objects that are not in `objmap`.

5.4.2 Restoring

The `restore()` method is the opposite of the saving methods. It takes the connection handle, class name, and identifier as its parameters. The identifier is the one that was passed
to the `save()` method during saving. It then calls the `deepRestore()` method to traverse through the object graph in the tables, instantiating objects as it goes along. The `ObjectCreator::create()` method is used to create a dummy object where the attributes in it are set to the restored values of the persisted object. If the said object has already been restored (present in `objmap`), then its attributes are overwritten with the restored values. This would mean that only one copy of the object can exist, even if `restore()` is called repeatedly for the same object. `restoremap` is used as a marker to prevent loops. The rest of the algorithm simply iterates through columns in the object’s record, traversing to the next table whenever it encounters a foreign key, otherwise it sets the object’s attribute to the persisted value. This method’s activity diagram is shown in Figure 5.8.
Figure 5.8 – Activity diagram for PersistenceBroker::restore(), deepRestore()

5.4.3 Deleting

Shown in Figure 5.9, the delete() method deletes persistent objects. It is called with the connection handle, class name, and identifier, similar to restore(). The algorithm uses refmap to keep track of reference counts. It first reads the record corresponding to the object and decrements its reference count and puts it in refmap. Then it recursively traverses through foreign keys only if the object referred to by the key is not already in refmap, meaning it has not been traversed. If it is present in refmap, then its reference count is decremented. After the object graph has been traversed completely, the actual deletion is done based on the entries in refmap. If an object’s reference count is zero, then it is deleted, otherwise it is updated with the new reference count. Thus, this ensures that in the case of overlapping object graphs, the overlapping objects are not deleted when one graph is deleted.
PersistenceBroker::deepDelete()

PersistenceBroker::delete()

Figure 5.9 – Activity diagram for PersistenceBroker::delete(), deepDelete()

5.4.4 Querying and Updating

If querying is supported by the underlying persistence mechanism (PersistenceMechanism::supportsSQL()), the user can call the query() method, passing
the connection handle and query statement. The query statement is in the form of a pseudo-SQL statement, adapted for simple object queries. As described in Section 4.2.2.5, the algorithm mainly involves parsing the pseudo-SQL statement, converting object references into table joins, and then passing the final translated SQL statement to PersistenceMechanism::query().

The update() method accepts input parameters comprising of the class name, attribute name, updated value, and search criteria for objects. If the underlying persistence mechanism supports queries (PersistenceMechanism::supportsSQL()), then a corresponding SQL statement is generated and passed to PersistenceMechanism::query(). The format and algorithm of the SQL is described in Section 4.2.2.6. If the persistence mechanism does not support SQL queries, then PersistenceMechanism::update() is called instead.

5.4.5 Persistence Mechanisms and Transactions

The broker also allows user-defined persistence mechanisms. registerMechanism() allows the user to register new mechanisms by just passing an identifier for the mechanism type (returned by PersistenceMechanism::getType()) and the class name of the persistence mechanism. It must implement the PersistenceMechanism interface to allow the broker to communicate with it. The broker will maintain a map (mech) of identifiers to the corresponding persistence mechanism. Therefore it can transparently support new mechanism types without any modification to its code. When support for a particular mechanism needs to be removed, the user can call deregisterMechanism(), passing the mechanism type as a parameter.

Transactions are supported if the persistence mechanism supports it (PersistenceMechanism::supportsTransactions()). Called with the connection handle as its parameter, startTransaction(), endTransaction(), and abortTransaction() simply call the corresponding PersistenceMechanism methods.

The transaction isolation level can also be set using setTransactionIsolation(). This determines the level of locking of the database during the transaction. setTransactionIsolation() retrieves the transaction isolation level, while supportsTransactionIsolationLevel() checks whether the database supports the specified
isolation level. These methods just directly call the corresponding PersistenceMechanism methods.

5.5 Framework facade

A facade can be defined as a packaged subset of components, or references to components, selected from the component system. Each facade provides public access to only those parts of the component system that have been chosen to be available for reuse [24]. In the context of a persistence framework, it forms the application programming interface for the user application as well as a component system for reusing the framework.

In the Framework facade, only three classes are visible, PersistenceBroker', PersistenceMechanism', and TableObj'. Under normal usage, the user application only needs to know the PersistenceBroker' interface for all persistence services. In the event that the framework functionality needs to be extended, all three classes can be extended to accommodate this. The user can implement additional persistence mechanisms by using PersistenceMechanism' and TableObj'.

A typical scenario is shown in Figure 5.10 whereby the framework is adapted for use in a patient information system. It inherits the persistence broker to implement a secure version that ensures data confidentiality. Support for various other persistence mechanisms are added where the patient data can be stored in an Oracle database or on the Internet through the HTTP and FTP protocols. This is in addition to the default persistence mechanisms supported by the framework (not visible in the facade).
Figure 5.10 – Reuse example of the persistence framework