4. Analysis

This chapter presents the use case model and analysis model of the framework. The use case model captures the requirements of the system while the analysis model describes the high-level design and object interactions.

4.1 Use Case Model

The use case model shown in Figure 4.1 consists of only two main actors, namely the user application and the database. The user application will initiate requests to the persistence framework which in turn interacts with the database where the data is to be stored. The database actor can be specialised into various different types of data stores, such as a relational, object, or flat file database. The description of the use cases are as follows:

- **Connect/Disconnect** – The user application passes the database name and address to the persistence framework, which then attempts to connect to and open the database. Similarly, the framework will disconnect from the database when the application requests it.
- **Save** – The user application passes an object and a unique object name to the persistence framework to be stored in a database. The framework then converts the object into a format compatible with the database and stores the object along with any other objects referenced by it.
- **Restore** – The user application passes a unique object name to the framework which serves as the object’s identifier. The framework then uses the identifier to retrieve the object from the database along with any other referenced objects. The object is then returned back to the user application.
- **Delete** – The user application passes the identifier of the object to be deleted to the framework. The framework will then access the database and delete the object and all objects referenced by it.
- **Query** – The user application passes a statement in a query language such as Structured Query Language (SQL) to the framework which then performs some appropriate translations and joins according to how the object is actually stored in the database. The statement is then submitted to the
database for execution and the results returned to the application.

- **Update** – The user application passes the class name, criteria on the object attributes, and the new value for a particular attribute. The framework will then look for objects of the specified class satisfying the criteria, and assigns a new value to the desired attribute.

- **Transaction** – The user application requests to either start, commit, or rollback a transaction. The user application can also set the transaction isolation level if required.

![Diagram](image)

**Figure 4.1** – Use case model of the persistence framework
4.2 Analysis model

Figure 4.2 shows the analysis model of the persistence framework. The analysis types can be grouped into four subsystems based on their functionality.

4.2.1 Object Mapper

The Object Mapper is a subsystem responsible for associating each persistent object with a unique and immutable identifier. Each Object is regarded as unique and no two objects can have the same Identifier. A global object table of all active persistent objects is maintained. When an Object is instantiated during a Restore, an Object-Identifier mapping will be created in the object table by the Map function. Similarly, when the Object is destroyed, the mapping will be removed. The core broker functions depend on the Object Mapper to enforce object identity.

4.2.2 Broker

The broker subsystem contains the functions described in the use case model. They are described in more detail in this section. The broker is responsible for performing the object-to-relational mappings to enable the object to be stored in the persistence mechanisms. Only concrete classes are stored, and each object and its inherited attributes are stored together as a single record in a table, as in the example shown in Figure 4.3 and Figure 4.4. Each concrete class is mapped to a single table, and each record in that table is an instance of the class. Relationships between classes are stored as foreign keys.

In Figure 4.3, the Employee class inherits attributes from the Person as well as the Human class. It also contains a reference to a Position class. When an Employee object is saved, all its attributes (EmployeeID, Position, Salary), including those of all its superclasses (Person, Human) in the inheritance hierarchy, is saved together into a single record in the Employee table. As the Position attribute refers to another object, a foreign key is stored which refers to the corresponding record in the Position table. Similarly, the Position table also contains a foreign key to the Department table. The mapped tables are shown in Figure 4.4. This mapping has the advantage of speed during queries because the data is stored in a structure natural to a relational database. However, the drawback is that it is difficult to
Figure 4.2 – Analysis model of the persistence framework
Figure 4.3 – Class hierarchy to be mapped

<table>
<thead>
<tr>
<th>Employee</th>
<th>Position</th>
<th>Salary</th>
<th>Name</th>
<th>Address</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>20069</td>
<td>69</td>
<td>1500</td>
<td>Ahmad</td>
<td>26, Jin 222</td>
<td>1.70</td>
<td>65</td>
</tr>
<tr>
<td>22011</td>
<td>24</td>
<td>5000</td>
<td>Jill Lim</td>
<td>12, Bugis St.</td>
<td>1.67</td>
<td>55</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4.4 – Mapped class hierarchy

<table>
<thead>
<tr>
<th>Position</th>
<th>Name</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>Programmer</td>
<td>27</td>
</tr>
<tr>
<td>24</td>
<td>Manager</td>
<td>11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>E-Commerce</td>
</tr>
<tr>
<td>11</td>
<td>Marketing</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
modify the class hierarchy in the database, especially with abstract base classes. Such modification would require all its concrete subclasses to be modified as well.

4.2.2.1 Connect/Disconnect

The User Application will pass the resource name, type, server address, username, and password to the broker for connecting to the Persistence Mechanism. The broker will choose an appropriate Persistence Mechanism class for the type of resource that it intends to connect to. The Connection class will store details of the connection, together with the Persistence Mechanism. The Connect function attempts to connect to the resource through the Persistence Mechanism using the specified username and password. If it is successful, it will return a handle to the user application. If a connection cannot be made after a certain timeout period, connection is refused, the resource is not available, or user authentication failed, then an error code is returned. Multiple simultaneous connections are supported. The Disconnect function performs the disconnection from a resource in a similar way. The sequence diagram is shown in Figure 4.5.

![Sequence diagram for the Connect/Disconnect use cases](image)

Figure 4.5 – Sequence diagram for the Connect/Disconnect use cases
4.2.2.2 Save

The user application will pass an Object to be saved and a connection handle to the broker. The Save function will Map the object and each referenced object to a unique Identifier. As its name suggests, the Identifier serves as a unique object identifier so that the object can be retrieved later. The Object is converted into a TableObj in relational form using the mappings described in Section 4.2.2. The Identifier is stored as the primary key of the TableObj. Simple as well as complex nested objects are supported and because of this, various different scenarios must be considered during the saving of the object. These are shown in Figure 4.6, which represents a group of objects and the references between them. Each box represents an object and each arrow represents an object reference. This diagram is commonly known as an object graph.

Figure 4.6 – Object graph to be saved. B is the persistent root object and shaded boxes represent all objects reachable by it.

Suppose that the user application passes object B to the broker to be saved. Therefore B is regarded as a persistent root object and all objects reachable by B are represented by shaded boxes. An object that is saved by the user is always considered a persistent root. To successfully save B, the Save function must save all objects reachable
by B as well. This involves traversing the object graph and saving each individual object until the whole graph is traversed. Objects that are not reachable, such as A and D, are not saved. Two special cases must be considered during saving:

1. **Multiple references to the same object** – This is illustrated by objects B and C both referencing the same object E. Therefore while traversing the object graph, E will be encountered twice, but it must be saved only once.

2. **Circular references** – This is shown by the two-way reference between objects C and E, as well as the larger reference loop B-E-C-F-G-B. Failure to recognize this will lead to an infinite loop in the graph traversal.

To solve this problem, each object would need to have a unique object identifier associated with it. When the object is saved, it is marked as saved if it has not already been saved. Therefore if the object is encountered again, it will be ignored and not saved again. This marking only lasts during the time of execution of the Save function and is cleared when the function returns. The **Identifier**, however, will last the lifetime of the object.

Figure 4.7 shows the sequence diagram.

![Sequence diagram for the Save use case](image)

**Figure 4.7** – Sequence diagram for the Save use case
4.2.2.3 Restore

To restore a persistent root object, the user application passes an Identifier to the broker. The user can only restore persistent root objects. The function then retrieves the record with the key matching the Identifier from the TableObj of the object’s class. From the data in the record, the corresponding Object is instantiated along with all other objects referenced by it. Referring again to Figure 4.6, when B is restored, the objects C, E, F, G, H, I will need to be instantiated as well as all the references between them. All Objects are then Mapped to their respective Identifiers in the Object Mapper.

As with saving, similar concerns need to be taken with multiple references and circular references, but the difference now is that tables and records need to be traversed rather than objects. The mechanism for solving this problem is quite similar — Identifiers of Objects that have been restored are marked as such and any attempt to restore the same Object is ignored. This will prevent instantiation of duplicates of the same object. Similar to the Save function, this marking only lasts until the function returns. If an object that is to be restored is already present in the Object Mapper, then it will be overwritten by the fresh copy from the persistence mechanism. Thus, repeated calls to Restore with the same Identifier will always return the same Object and not generate duplicates. The restoring process is shown as a sequence diagram in Figure 4.8.

![Sequence diagram for the Restore use case](image-url)
4.2.2.4 Delete

The **Delete** function, when called by the user application, removes the persistent root object and all objects reachable by it. This requires traversing through tables and deleting records from it. When an object is deleted, its reference count is decreased by one. A persistent object will only be removed when there are no references to it, that is, its reference count is zero. Again, a temporary marking is used to prevent loops, but in the case of the object having multiple references, its reference count is decreased for every time it is encountered during traversal. The sequence diagram is shown in Figure 4.9

![Sequence diagram for the Delete use case](image)

**Figure 4.9** – Sequence diagram for the Delete use case

4.2.2.5 Query

This allows the user application to perform queries on persistent objects that have been saved. Typically, a query statement is passed to the broker, which then **Maps** it into a suitable SQL statement to be executed by the persistence mechanism. Some translations need to be made because of the differences in the way object attributes are addressed compared to columns in relational tables. In a regular SQL statement, a query would
typically be performed as “SELECT [column] FROM [table] WHERE [condition]”. This will produce a result set of records containing the specified columns from the specified tables that satisfy the specified conditions. The column and table names used here are just simple names, but in an object, the addressing of its attributes can be very much different due to the possibility of nested objects. As an example, consider the following object \(b\) of class \(B\) (shown in Java/C++ language):

```java
class A {
    int attr1;
    int attr2;
};

class B {
    int attr1;
    A a;
};
```

Consider the case where \(b\) is saved as a persistent root. Therefore, when the user application submits a query, it will need to be able to address the attributes contained in \(b\) as well as those in the nested object \(a\). The notation is just to use a dot \(\cdot\) as a separator, for example, \(b\cdot attr1\), \(b.a.attr1\), and \(b.a.attr2\). As such, names and conditions can be expressed easily as a pseudo-SQL statement such as “SELECT * FROM B WHERE a.attr1=1;”. This will produce a result set of all objects of class \(B\) whose attribute \(a.attr1\) is equal to 1. Since this pseudo-SQL statement needs to be translated to a valid SQL statement, the \texttt{Query} function itself will be responsible for the translation, depending on the way the objects are mapped to the relational schema. For every nested object, there will need to be a table join, so the actual SQL statement to be executed would be “SELECT * FROM B, A WHERE B.a=A.obj_id AND A.attr1=1;”, represented by \texttt{Query Statement}. The \texttt{obj_id} here is the unique identifier \texttt{Identifier} that was associated with \(a\) when it was saved. From the result table, the corresponding objects of class \(B\) will be instantiated and returned as an object array where each object in the array satisfies the query criteria. The \texttt{Query} sequence diagram is shown in Figure 4.10.
4.2.2.6 Update

Shown in Figure 4.11, this function allows the user application to modify the attributes of persistent objects that have been saved. The input data is comprised of the class name, attribute name, updated value, and search criteria. These are then mapped into a **Query Statement** which contains SQL statements to perform the update. This is then passed to the **Persistence Mechanism** where it is executed. As with **Query**, certain translations have to be performed when the user addresses nested object attributes or sets an attribute to a non-primitive object value. As in the example class diagram in Section 4.2.2.5, if the user requests to set the value of B.a.attr2=6 which satisfies the condition B.attr1=1, then the corresponding pseudo-SQL statement is "UPDATE B SET B.a.attr2=6 where B.attr1=1;". Therefore the translation would involve a table join between A and B, giving the final translated statement as "UPDATE A,B SET A.attr2=6 where B.a=A.obj_id AND B.attr1=1;" where obj_id is the unique identifier **Identifier** that was associated with a when it was saved.

In another use case scenario (not shown in Figure 4.11), suppose the user creates a
new object a1 of class A and wishes to set B.a=a1 with the condition that B.attr1=1. Since this involves setting objects as attributes, there is no corresponding SQL statement as SQL only allows simple types as values. Therefore the **Update** function must first **Save** the new object a1, obtain its **Identifier**, and set the foreign key in B to this identifier. For example, if the identifier is "A@216bdf", then the SQL statement "UPDATE B SET a='A@216bdf' WHERE attr1=1;" is executed.

4.2.3 Relational Data

This subsystem contains all the data structures necessary when saving data in relational form. The **TableObj** type is an class that wraps around relational table data, which consists of records and columns. This class contains methods to search, add, change and retrieve individual records as well as the fields in those records. An **Object** will typically be mapped to records in one or more **TableObj**s where each **TableObj** contains objects of a particular class. The **Query Statement** encapsulates all queries to the database while **Transaction** represents a group of **Query Statements** that must be executed or rolled back as an atomic unit.
4.2.4 Persistence Mechanism

This subsystem encapsulates the persistent storage mechanism. It handles all details specific to the storage mechanism, such as the data format, type conversion, establishing connections, and reading and writing data. The Persistence Mechanism accepts TableObj data and saves it in its native data format, and vice-versa for restoring data. It contains methods to create, read, update, and delete tables from the persistence mechanism. During creation of a table, it must read data from the TableObj, convert the data types from programming language-specific types to its native data type formats, and save it into a table. In reading, the reverse process occurs. The update method allows for an existing table to be updated with new data, while the delete method deletes a table from the persistence mechanism. The Persistence Mechanism also contains methods to handle transactions, transaction isolation, and execute queries. Each connection to the persistence mechanism is represented by a Connection. The object-relational mappings and any object-related data are invisible to this subsystem as it only deals with data in the standard relational table form.