Implementing an object-oriented design generally involves using a standard object oriented programming language OOPL or mapping object designs to databases. In most cases, it involves both.

**Implementation using Programming Languages**

Usually, the task of transforming an object design into code is a straightforward process. Any object-oriented programming language like C++, Java, Smalltalk, C# and Python, includes provision for representing classes. In this chapter, we exemplify the concept using C++.

The following figure shows the representation of the class Circle using C++.

![Class Circle representation using C++](image)

**Implementing Associations**

Most programming languages do not provide constructs to implement associations directly. So the task of implementing associations needs considerable thought.

Associations may be either unidirectional or bidirectional. Besides, each association may be either one-to-one, one-to-many, or many-to-many.

**Unidirectional Associations**

For implementing unidirectional associations, care should be taken so that unidirectionality is maintained. The implementations for different multiplicity are as follows:

- **Optional Associations**: Here, a link may or may not exist between the participating objects. For example, in the association between Customer and Current Account in the figure below, a customer may or may not have a current account.

![Association example](image)

For implementation, an object of Current Account is included as an attribute in Customer that may be NULL. Implementation using C++:

```cpp
class Customer
{
    private:
        // attributes
        Current_Account c; //an object of Current_Account as attribute

    public:
        // methods
};
```
Customer()
{
  c = NULL;
} // assign c as NULL

Current_Account getCurra()
{
  return c;
}

void setCurra( Current_Account myacc)
{
  c = myacc;
}

void removeAcc()
{
  c = NULL;
}

• **One-to-one Associations**: Here, one instance of a class is related to exactly one instance of the associated class. For example, Department and Manager have one-to-one association as shown in the figure below.

```
Department -- 1 --> Manager
```

This is implemented by including in Department, an object of Manager that should not be NULL. Implementation using C++:

```cpp
class Department
{
  private:
  // attributes
  Manager mgr; // an object of Manager as attribute

  public:
  Department /*parameters*/ , Manager m) //m is not NULL
  {
    // assign parameters to variables
    mgr = m;
  }

  Manager getMgr()
  {
    return mgr;
  }
};
```

• **One-to-many Associations**: Here, one instance of a class is related to more than one instances of the associated class. For example, consider the association between Employee and Dependent in the following figure.

```
Employee -- 0...* --> Dependent
```

This is implemented by including a list of Dependents in class Employee. Implementation using C++ STL list container:

```cpp
class Employee
{
  private:
  char * deptName;
  list <Dependent> dep; // a list of Dependents as attribute
};
```
public:
  void addDependent ( Dependent d)
  {
    dep.push_back(d);
    // adds an employee to the department
  }
  void removeDependent( Dependent d)
  {
    int index = find ( d, dep );
    // find() function returns the index of d in list dep
    dep.erase(index);
  }
};

Bi-directional Associations

To implement bi-directional association, links in both directions require to be maintained.

- **Optional or one-to-one Associations**: Consider the relationship between Project and Project Manager having one-to-one bidirectional association as shown in the figure below.

![Bi-directional Association Diagram](image)

Implementation using C++:

```cpp
Class Project
{
  private:
    // attributes
    Project_Manager pmgr;
  public:
    void setManager ( Project_Manager pm);
    Project_Manager changeManager()
};
class Project_Manager
{
  private:
    // attributes
    Project pj;
    public:
    void setProject(Project p);
    Project removeProject();
};
```

- **One-to-many Associations**: Consider the relationship between Department and Employee having one-to-many association as shown in the figure below.

![One-to-many Association Diagram](image)

Implementation using C++ STL list container:

```cpp
class Department
{
  private:
    char * deptName;
    list <Employee> emp; // a list of Employees as attribute
  public:
    void addEmployee ( Employee e)
    {
      emp.push_back(e);
    }
};
```
void removeEmployee(Employee e) {
    int index = find(e, emp); // find function returns the index of e in list emp
    emp.erase(index);
}

class Employee {
    private:
        //attributes
        Department d;
    public:
        void addDept();
        void removeDept();
};

Implementing Associations as Classes

If an association has some attributes associated, it should be implemented using a separate class. For example, consider the one-to-one association between Employee and Project as shown in the figure below.

![Diagram of Employee and Project association](image)

Implementation of WorksOn using C++:

class WorksOn {
    private:
        Employee e;
        Project p;
        Hours h;
        char * date;
    public:
        // class methods
};

Implementing Constraints

Constraints in classes restrict the range and type of values that the attributes may take. In order to implement constraints, a valid default value is assigned to the attribute when an object is instantiated from the class. Whenever the value is changed at runtime, it is checked whether the value is valid or not. An invalid value may be handled by an exception handling routine or other methods.

Example

Consider an Employee class where age is an attribute that may have values in the range of 18 to 60. The following C++ code incorporates it:

class Employee {
    private: char * name;
}
```cpp
int age;
// other attributes

public:
Employee() // default constructor
{
    strcpy(name, "")
    age = 18; // default value
}

class AgeError {}; // Exception class
void changeAge(int a) // method that changes age
{
    if (a < 18 || a > 60) // check for invalid condition
        throw AgeError(); // throw exception
    age = a;
}

Implementing State Charts

There are two alternative implementation strategies to implement states in state chart diagrams.

Enumerations within Class

In this approach, the states are represented by different values of a data member `setofofdatamembers`. The values are explicitly defined by an enumeration within the class. The transitions are represented by member functions that change the value of the concerned data member.

Arrangement of Classes in a Generalization Hierarchy

In this approach, the states are arranged in a generalization hierarchy in a manner that they can be referred by a common pointer variable. The following figure shows a transformation from state chart diagram to a generalization hierarchy.

Object Mapping to Database System

Persistency of Objects

An important aspect of developing object-oriented systems is persistency of data. Through persistency, objects have longer lifespan than the program that created it. Persistent data is saved on secondary storage medium from where it can be reloaded when required.

Overview of RDBMS

A database is an ordered collection of related data.

A database management system `DBMS` is a collection of software that facilitates the processes of defining, creating, storing, manipulating, retrieving, sharing, and removing data in databases.
In relational database management systems (RDBMS), data is stored as relations or tables, where each column or field represents an attribute and each row or tuple represents a record of an instance.

Each row is uniquely identified by a chosen set of minimal attributes called **primary key**.

A **foreign key** is an attribute that is the primary key of a related table.

### Representing Classes as Tables in RDBMS

To map a class to a database table, each attribute is represented as a field in the table. Either an existing attribute is assigned as a primary key or a separate ID field is added as a primary key. The class may be partitioned horizontally or vertically as per requirement.

For example, the Circle class can be converted to table as shown in the figure below.

```
<table>
<thead>
<tr>
<th>Circle</th>
<th>C_id</th>
<th>x-coord</th>
<th>y-coord</th>
<th>radius</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C1</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>-5</td>
<td>7</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>8</td>
<td>-8</td>
<td>10</td>
<td>yellow</td>
</tr>
</tbody>
</table>
```

**Schema for Circle Table:** CIRCLE(CID, XCOORD, YCOORD, RADIUS, COLOR)

**Creating a Table Circle using SQL command:**

```sql
CREATE TABLE CIRCLE
(
  CID VARCHAR2(4) PRIMARY KEY,
  XCOORD INTEGER NOT NULL,
  YCOORD INTEGER NOT NULL,
  ZCOORD INTEGER NOT NULL,
  COLOR
);```

### Mapping Associations to Database Tables

#### One-to-One Associations

To implement 1:1 associations, the primary key of any one table is assigned as the foreign key of the other table. For example, consider the association between Department and Manager:

```
<table>
<thead>
<tr>
<th>Department</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dep_ID</td>
<td>EmpID</td>
</tr>
<tr>
<td>DName</td>
<td>EName</td>
</tr>
<tr>
<td>Location</td>
<td>Address</td>
</tr>
</tbody>
</table>
```

SQL commands to create the tables:

```sql
CREATE TABLE DEPARTMENT
(
  DEPT_ID INTEGER PRIMARY KEY,
  DNAME VARCHAR2(30) NOT NULL,
  LOCATION VARCHAR2(20),
  EMPID INTEGER REFERENCES MANAGER
);

CREATE TABLE MANAGER
(
```

```
One-to-Many Associations

To implement 1:N associations, the primary key of the table in the 1-side of the association is assigned as the foreign key of the table at the N-side of the association. For example, consider the association between Department and Employee:

<table>
<thead>
<tr>
<th>Department</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dep_ID</td>
<td>EmpID</td>
</tr>
<tr>
<td>DName</td>
<td>EName</td>
</tr>
<tr>
<td>Location</td>
<td>Address</td>
</tr>
</tbody>
</table>

SQL commands to create the tables:

```sql
CREATE TABLE DEPARTMENT
(
  DEPT_ID INTEGER PRIMARY KEY,
  DNAME VARCHAR2(30) NOT NULL,
  LOCATION VARCHAR2(20),
);

CREATE TABLE EMPLOYEE
(
  EMPID INTEGER PRIMARY KEY,
  ENAME VARCHAR2(50) NOT NULL,
  ADDRESS VARCHAR2(70),
  D_ID INTEGER REFERENCES DEPARTMENT
);
```

Many-to-Many Associations

To implement M:N associations, a new relation is created that represents the association. For example, consider the following association between Employee and Project:

<table>
<thead>
<tr>
<th>Employee</th>
<th>*</th>
<th>*</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Schema for Works_On Table**: WORKS_ON EMPID, PID, HOURS, START_DATE

**SQL command to create Works_On association**: CREATE TABLE WORKS_ON

```sql
CREATE TABLE WORKS_ON
(
  EMPID INTEGER,
  PID INTEGER,
  HOURS INTEGER,
  START_DATE DATE,
  PRIMARY KEY (EMPID, PID),
  FOREIGN KEY (EMPID) REFERENCES EMPLOYEE,
  FOREIGN KEY (PID) REFERENCES PROJECT
);
```
Mapping Inheritance to Tables

To map inheritance, the primary key of the base tables is assigned as the primary key as well as the foreign key in the derived tables.

Example

```
SCHEMAS:
EMPLOYEE (EID, ENAME)
MANAGER (EID, TITLE, DEPARTMENT)
   (EID references EMPLOYEE)
SCIENTIST (EID, PUBS, SPCL)
   (EID references EMPLOYEE)
```