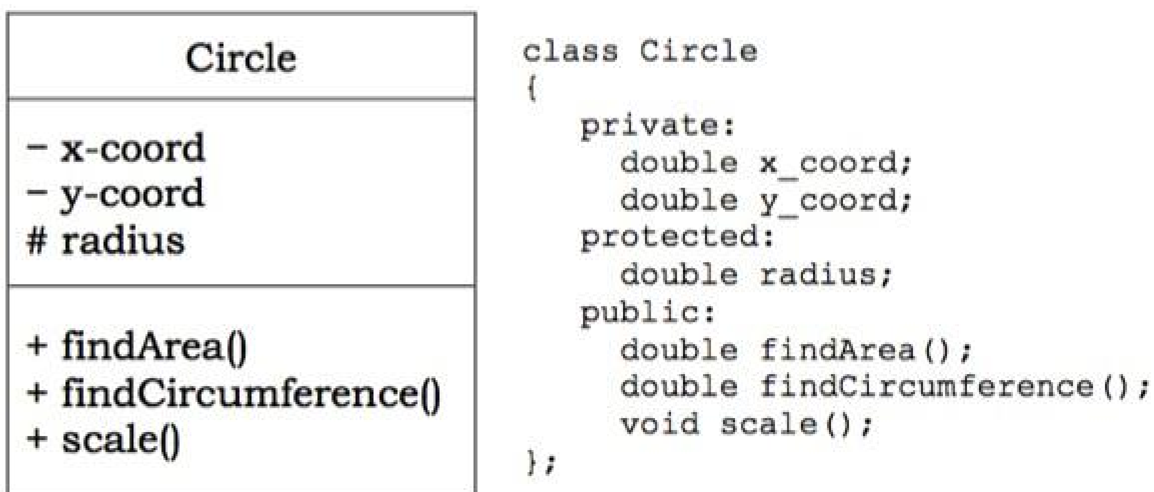
OOAD - Implementation Strategies

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++.



Implementing Associations

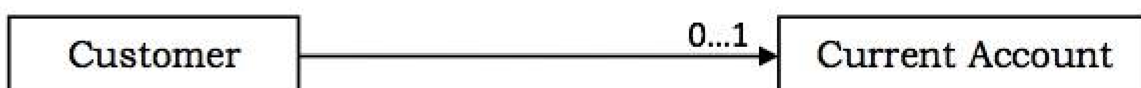
Most programming languages do not provide constructs to implement associations directly. So the task of implementing associations needs considerable thought.

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.



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

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

   public:

   Customer() {
      c = NULL;
   } // assign c as NULL

   Current_Account getCurrAc() {
      return c;
   }

   void setCurrAc( 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.

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

```
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.

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

```
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 removeDeoendent( 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.

Implementation using C++ −

```
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();
};
```

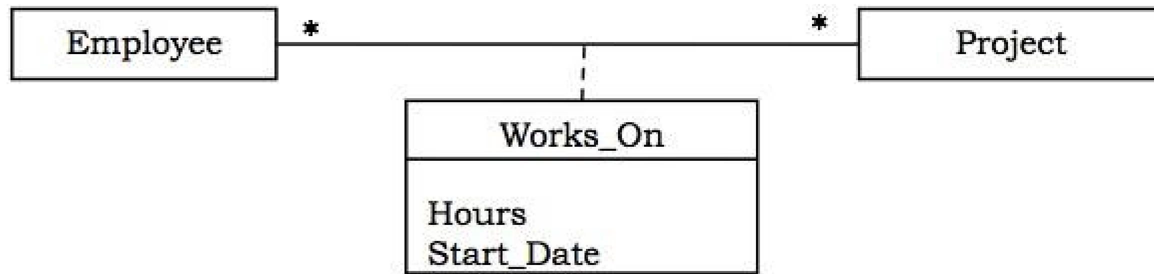
- **One–to–many Associations** − Consider the relationship between Department and Employee having one–to–many association as shown in the figure below.

Implementation using C++ STL list container

```
class Department {
   private:
   char * deptName;
   list <Employee> emp; //a list of Employees as attribute

   public:
   void addEmployee ( Employee e) {
      emp.push_back(e);
   } // adds an employee to the department

   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.



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;
   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;
   }
};
```