18CSC202J - Syllabus

Unit 3 :

Single and Multiple Inheritance - Multilevel inheritance - Hierarchical - Hybrid Inheritance - Advanced Functions - Inline - Friend - Virtual -Overriding - Pure virtual function - Abstract class and Interface -UML State Chart Diagram - UML Activity Diagram

Types of Inheritance

Inheritance

01 Definition

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is

02 Syntax

```
class Subclass_name : access_mode Superclass_name
```

03 Types

Single Inheritance, Multiple Inheritance, Hierarchical Inheritance, Multilevel Inheritance, and Hybrid Inheritance (also known as Virtual Inheritance)

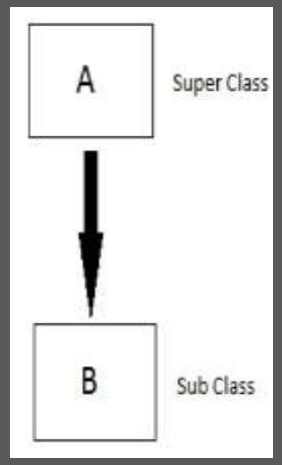
Note : All members of a class except Private, are inherited

04 Advantages

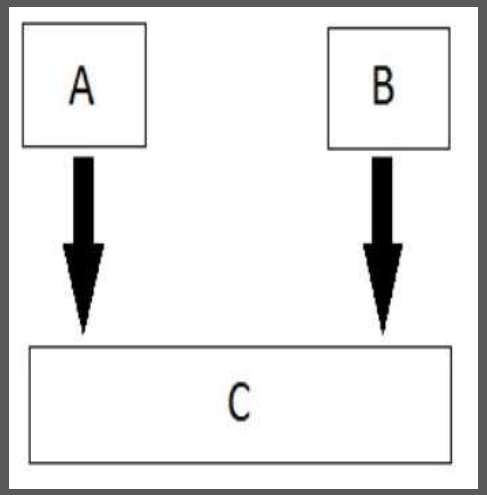
1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

Inheritance Types

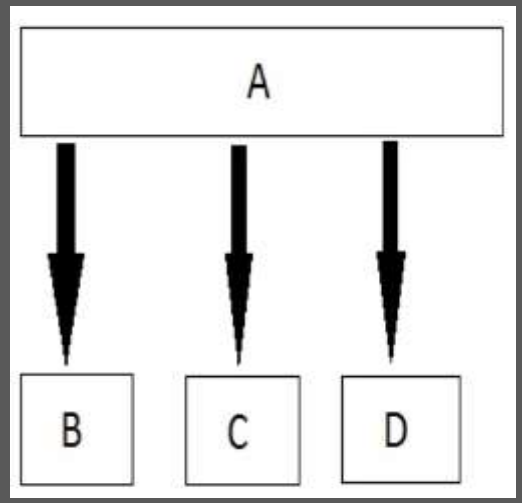
01 Single



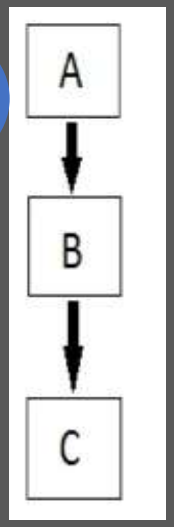
02 Multiple



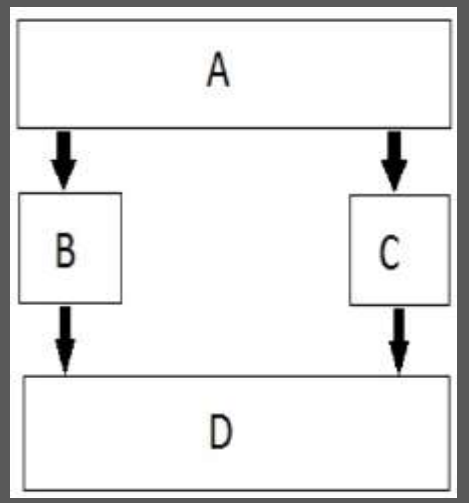
03 Hierarchical



04 Multilevel



05 Hybrid



Modes of Inheritance

01 Public

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class

02 Protected

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

03 private

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

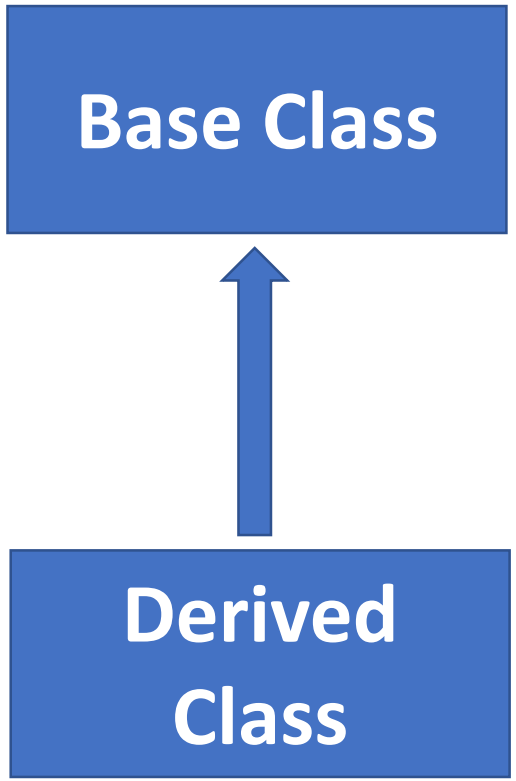
Note:

The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example

Inheritance Access Matrix

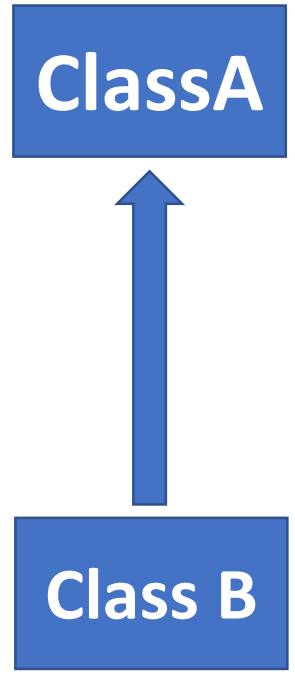
Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

SINGLE INHERITANCE



Example

Base Class



Derived Class

Single Inheritance

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only. Based on the visibility mode used or access specifier used while deriving, the properties of the base class are derived. Access specifier can be private, protected or public.

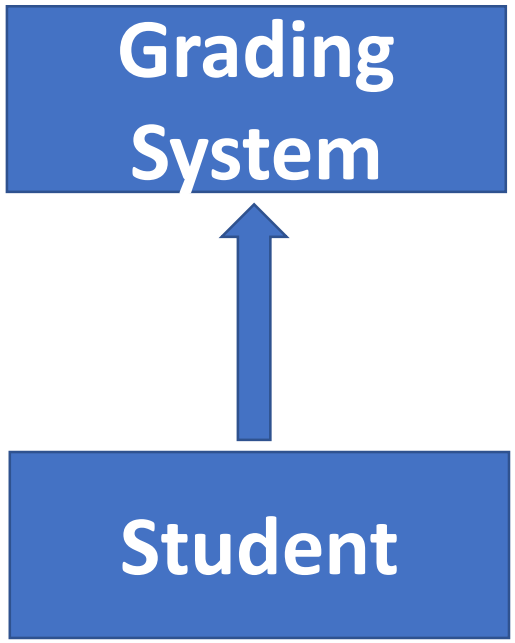
Syntax:

```
class Classname   // base class
{
    ..........
};
class classname: access_specifier baseclassname
{
...
};
```

Example

```
#include <iostream>
using namespace std;
class base    //single base class
{  public:
     int x;
       void getdata()
       {
        cout << "Enter the value of x = ";
        cin >> x;
        }
 };
class derived : public base    //single derived
{
    int y;
   public:
      void readdata()
      {
     cout << "Enter the value of y = ";
     cin >> y;
      }
   void product()
   {
       cout << "Product = " << x * y;
    }
};

    int main()
    {
        derived a;   //object of derived class

         a.getdata();

        a.readdata();

        a.product();

        return 0;
      }
```

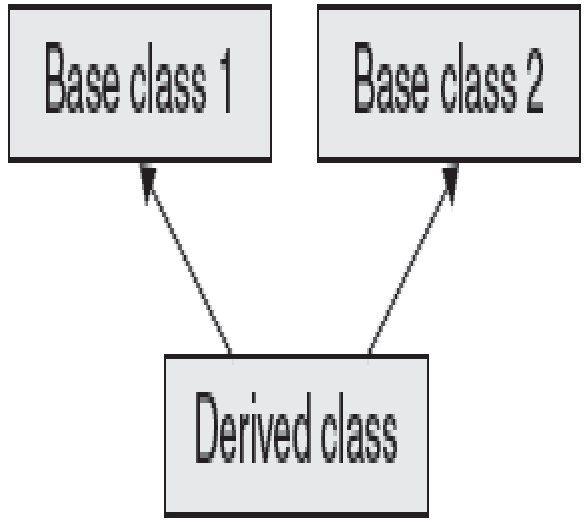
Applications of Single Inheritance



1. University Grading System
2. Employee and Salary

Multiple Inheritance

Base Class



Base Class

Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.

Syntax:

```
class A   // base class
{
    ..........
};
class B
{
    ..........
}
class c : access_specifier A, access_specifier B   // derived class
{
    ..........
} ;
```

Example:

```
#include using namespace std;
class sum1
{
        protected: int n1;
};
class sum2
{
        protected: int n2;
};
class show : public sum1, public
sum2
{
public: int total()
{
cout<<“enter n1”;
cin>>n1;
cout<<“enter n2”;
cin>>n2;
cout<<“sum=”<
cout<<“sum=”<<n1+n2<<endl;
}
};
int main()
{
show ob;
ob.total();
}
```

Applications of Multiple Inhertiance

- Distributed Database

Multilevel Inheritance

A derived class can be derived from another derived class. A child class can be the parent of another class.

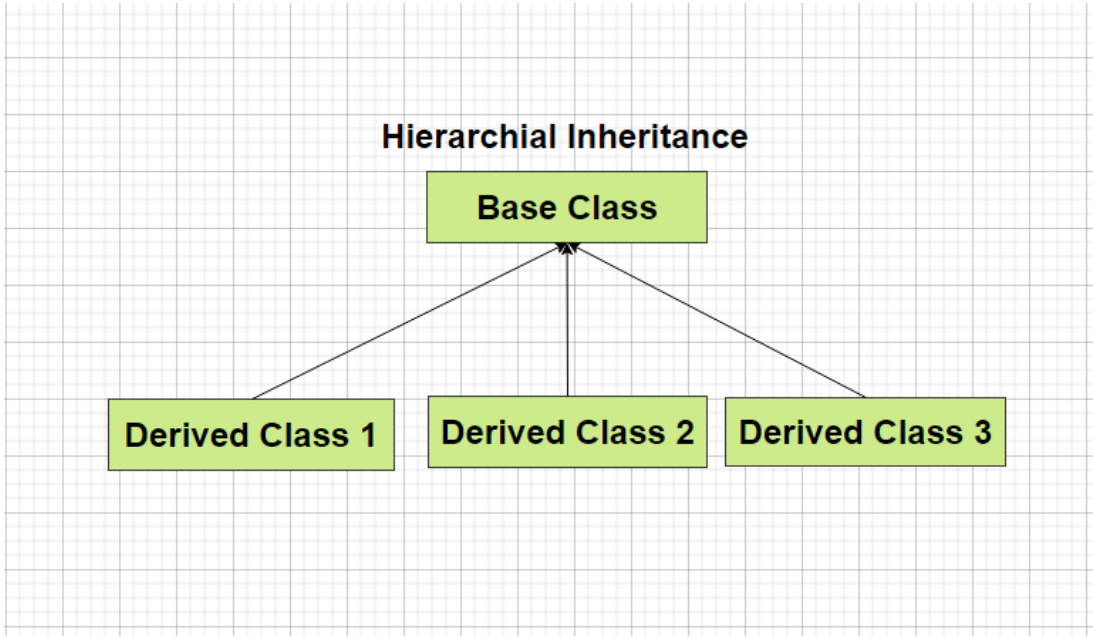
Syntax:

```
class A   // base class
{
   ..........
};
class B
{
    ..........
}
class C : access_specifier B
// derived class
{
   ...........
} ;
```

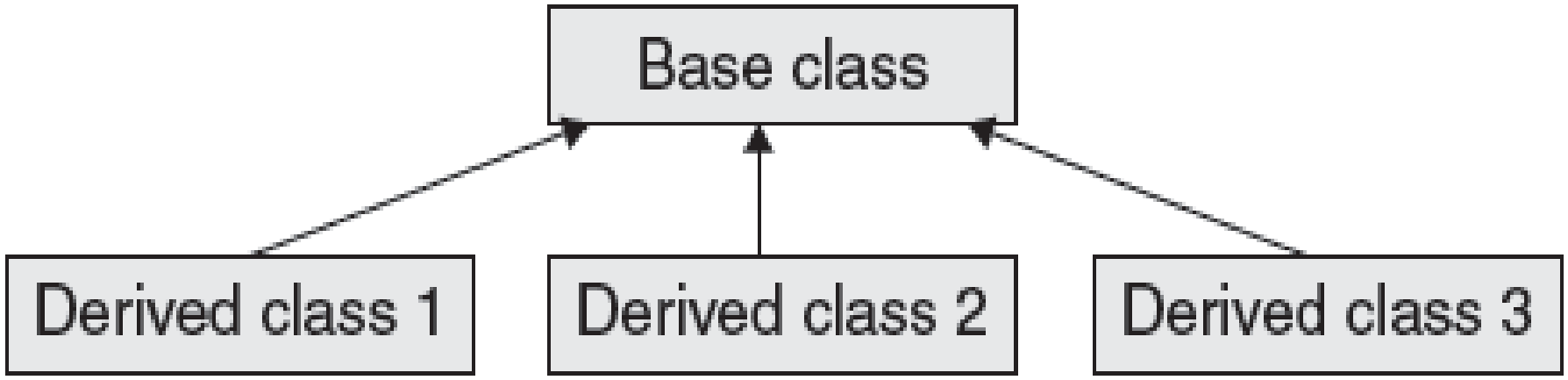
```
// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle";
    }
};
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are
vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
  public:
    car()
    {
      cout<<"Car has 4 Wheels”;
    }
};
  // main function
int main()
{
//creating object of sub class will
  //invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Hierarichal Inheritance

- In Hierarichal Inheritance we have several classes that are derived from a common base class (or parent class).
- Here in the diagram Class 1, Class 2 and Class 3 are derived from a common parent class called Base Class.



Hierarchical Inheritance



How to implement Hierarchal Inheritance in C++

```
class A {
 // Body of Class A
};  // Base Class

class B : access_specifier A
{
 // Body of Class B
}; // Derived Class

class C : access_specifier A
{
 // Body of Class C
}; // Derived Class
```

- In the example present in the left we have class A as a parent class and class B and class C that inherits some property of Class A.
- While performing inheritance it is necessary to specify the access_specifier which can be public, private or protected.

Example

```
#include <iostream>
using namespace std;
class A //single base class
{
    public:
            int x, y;
            void getdata()
            {
           cout << "\nEnter value of x and y:\n";
           cin >> x >> y;
            }
};
class B : public A //B is derived from class base
{
   public:
            void product()
            {
           cout << "\nProduct= " << x * y;
            }
};
```

```
class C : public A //C is also derived from
class base
{
   public:
            void sum()
            {
      cout << "\nSum= " << x + y;
            }
};
int main()
{
   B obj1;       //object of derived class B
   C obj2;       //object of derived class C
   obj1.getdata();
   obj1.product();
   obj2.getdata();
   obj2.sum();
   return 0;
}
```

Example of **Hierarchical Inheritance**

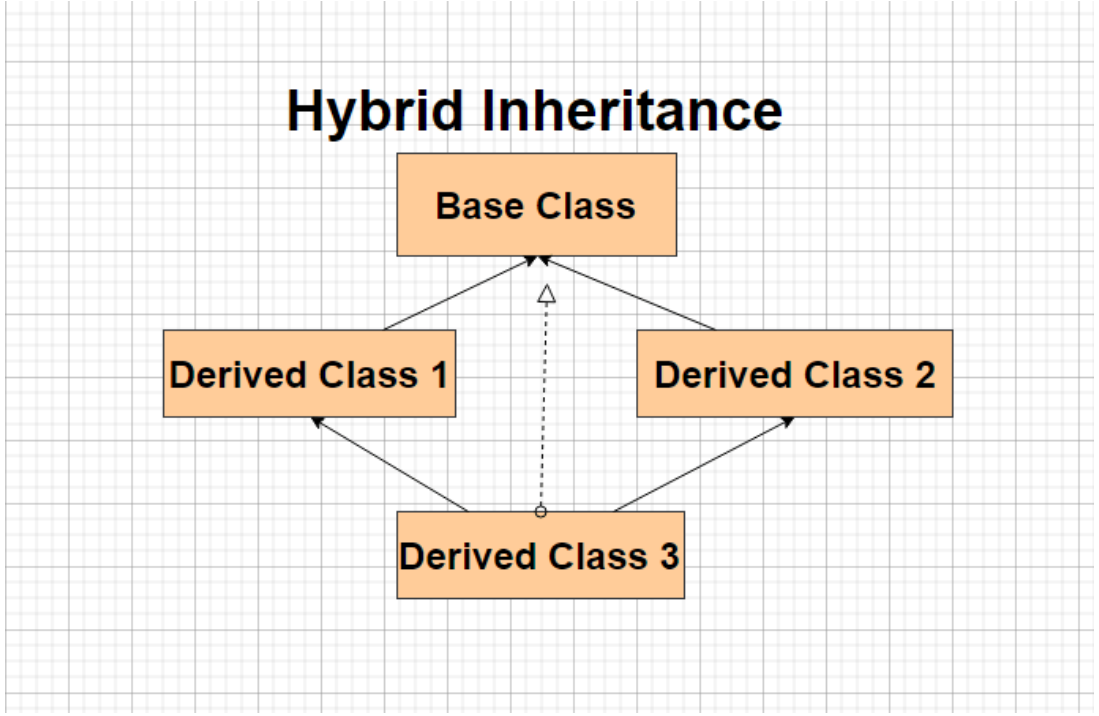
```
class Car {
  public:
    int wheels = 4;
    void show() {
      cout << "No of wheels : "<<wheels ;
    }
};

class Audi: public Vehicle {
  public:
    string brand = "Audi";
    string model = “A6”;
};
```

```
class Volkswagen: public Car {
  public:
    string brand = "Volkswagen";
    string model = “Beetle”;
};

class Honda: public Car {
  public:
    string brand = "Honda";
    string model = “City”;
};
```

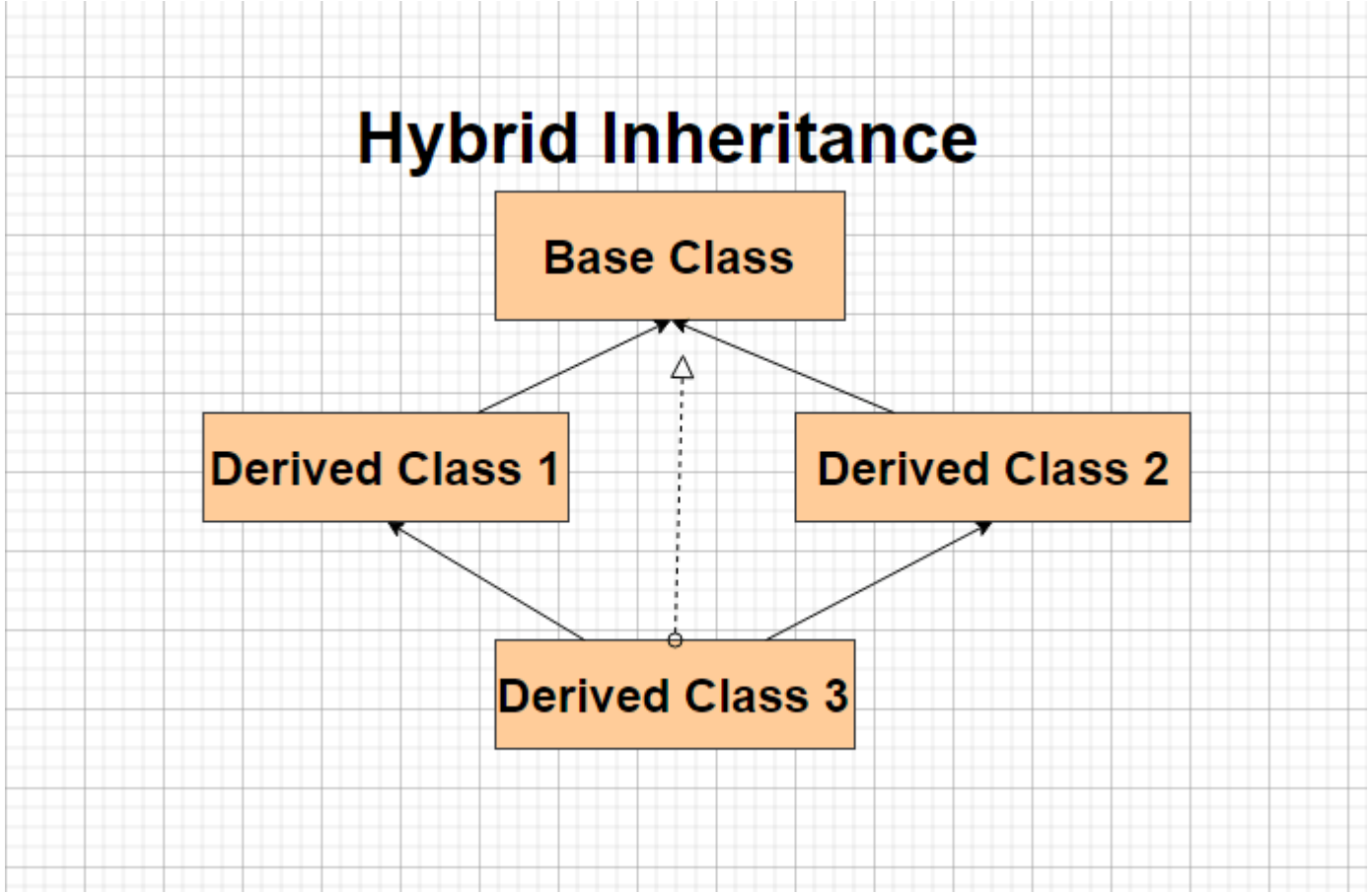
Hybrid Inheritance

Hybrid Inheritance

- Hybrid Inheritance involves derivation of more than one type of inheritance.
- Like in the given image we have a combination of hierarichal and multiple inheritance.
- Likewise we can have various combinations.



Diagramatic Representation of Hybrid Inheritance



How to implement Hybrid Inheritance in C++

```
class A
{
  // Class A body
};

class B : public A
{
    // Class B body
} ;

class C
{
  // Class C body
};
```

- Hybrid Inheritance is no different than other type of inheritance.
- You have to specify the access specifier and the parent class in front of the derived class to implement hybrid inheritance.

Access Specifiers

In C++ we have basically three types of access specifiers :

- Public : Here members of the class are accessible outside the class as well.
- Private : Here members of the class are not accessible outside the class.
- Protected : Here the members cannot be accessed outside the class, but can be accessed in inherited classes.

Example of Hybrid Inheritance

```
class A
{
        public:
        int x;
};

class B : public A
{
        public:
        B()
        {
          x = 10;
        }
};
```

```
class C
{       public:
        int y;
        C()
        {
            y = 4;
      }
};
class D : public B, public C
{       public:
        void sum()
        {
            cout << "Sum= " << x + y;
        }
```

Order of Constructor Call

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Points to Remember

- Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.

- To call base class's parameterised constructor inside derived class's parameterised constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Example

```
class Base
{
   int x;
   public:
Base()
   {
cout<<"Base default constructor";
   }
};

class Derived : public Base
{
   int y;
   public:
Derived()
   {
cout<"Derived def. constructor";
   }
   // parameterized constructor
   Derived(int i)
   {
   cout << "Derived parameterized
constructor\n";
   }
};

int main()
{
   Base b;
   Derived d1;
   Derived d2(10);
}
```

Order of Constructor Call

Example:

```
class Base
{
   int x;
   public:
   // parameterized constructor
   Base(int i)
   {
      x = i;
      cout<<"BaseParameterized Constructor\n";
   }
};

class Derived : public Base
{
   int y;
   public:
   // parameterized constructor
Derived(int j):Base(j)
   {
      y = j;
cout << "Derived Parameterized Constructor\n";
   }
};

int main()
{
   Derived d(10) ;
}
```

Note:

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

01 Concept

- Constructors and Destructors are never inherited and hence never overridden.
- Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class.

02 Static Function

- They are inherited into the derived class.
- If you redefine a static member function in derived class, all the other overloaded functions in base class are hidden.
- Static Member functions can never be virtual.

03 Limitation

Derived class can inherit all base class methods except:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Calling base and derived class method using base reference

Example:

```
#include <iostream>
using namespace std;

class Foo
{
public:
   int x;

   virtual void printStuff()
   {
 cout<<"BaseFoo
printStuff called"<<endl;
   }
};

class Bar : public Foo
{
public:
   int y;
   void printStuff()
   {
      cout<<"derived      Bar      printStuff
called"<<endl;
   }
};

int main()
{

   Foo *foo=new Foo;
   foo->printStuff();/////this call the base
function
   foo=new Bar;
   foo->printStuff();
}
```

Output:

Base Foo printStuff called
derived Bar printStuff called

Calling base and derived class method using derived reference

Example :

```
#include <iostream>

class Base{
  public:
    void foo()
          {
                std::cout<<"base";
          }
};
class Derived : public Base
{
  public:
    void foo()
          {
                std::cout<<"derived";
          }
};

int main()
{
  Derived bar;
  //call Base::foo() from bar here?
  bar.Base::foo(); // using a qualified-id
  return 0;
}
```

Output:

base

Advanced Functions: Inline, Friend, Virtual function and Overriding

Friend Function

1. The main concepts of the object oriented programming paradigm are data hiding and data encapsulation.

2. Whenever data variables are declared in a private category of a class, these members are restricted from accessing by non – member functions.

3. The private data values can be neither read nor written by non – member functions.

4. If any attempt is made directly to access these members, the compiler will display an error message as “inaccessible data type”.

5. The best way to access a private data member by a non – member function is to change a private data member to a public group.

6. When the private or protected data member is changed to a public category, it violates the whole concept or data hiding and data encapsulation.

7. To solve this problem, a friend function can be declared to have access to these data members.

8. Friend is a special mechanism for letting non – member functions access private data.

9. The keyword friend inform the compiler that it is not a member function of the class.

Friend Function

Granting Friendship to another Class

1. A class can have friendship with another class.

2. For Example, let there be two classes, first and second. If the class first grants its friendship with the other class second, then the private data members of the class first are permitted to be accessed by the public members of the class second. But on the other hand, the public member functions of the class first cannot access the private members of the class second.

01

Syntax

```
class second;        forward declaration

class first

        {

                     private:

                              ----------
```

Friend Function

Two classes having the same Friend

1. A non – member function may have friendship with one or more classes.

2. When a function has declared to have friendship with more than one class, the friend classes should have forward declaration.

3. It implies that it needs to access the private members of both classes.

01

Syntax

friend return_type function_name(parameters);

02

Example

friend return_type fname(first one, second two)
{}

03

Note:

where friend is a keyword used as a function modifier. A friend declaration is valid only within or outside the class definition.

Friend Function

Syntax:

```
class second;       forward declaration
        class first
        {
                private:
                        --------------
                public:
                        friend return_type
fname(first one, second two);
        };
        class second
        {
                private:
                        -----------------
                public:
                        friend return_type
fname(first one, second two);
        };
```

Case 2:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual void display();
                virtual static int sum();      //error
}
int sample::sum()
{ }
```

Friend Function Example

Example:

```
class sample
{
        private:
                int x;
        public:
                void getdata();
                friend void display(sample abc);
};
void sample::getdata()
{
        cout<<"Enter a value for x\n"<<endl;
        cin>>x;
}
void display(sample abc)
{
        cout<<"Entered Number is  "<<abc.x<<endl;
}
```

```
void main()
{
        clrscr();

        sample obj;

        obj.getdata();
        cout<<"Accessing the private data by non - member function"<<endl;
        display(obj);

        getch();
}*/
```

Friend Function Example

Example:

```
class first
{
        friend class second;
        private:
                int x;
        public:
                void getdata();
};

class second
{
        public:
                void disp(first temp);
};

void first::getdata()
{
        cout<<"Enter a Number ?"<<endl;
        cin>>x;
}

void second::disp(first temp)
{
        cout<<"Entered Number is = "<<temp.x<<endl;
}

void main()
{

        first objx;
        second objy;
        objx.getdata();
        objy.disp(objx);
}
```

Friend Function Example

```
class second;      //Forward
Declaration
class first
{
        private:
                int x;
        public:
void getdata();
void display();
friend int sum(first one,second two);
};
class second
{
        private:
                int y;
        public:
                void getdata();
                void display();
friend int sum(first one,second two);
};
void first::getdata()
{
cout<<"Enter a Value for X"<<endl;
                cin>>x;
}
void second::getdata()
{
cout<<"Enter a value for Y"<<endl;
                cin>>y;
}
void first::display()
{
cout<<"Entered Number is X = ";
}
void second::display()
{
cout<<"Entered Number is Y = ";
}
int sum (first one,second two)
{
        int temp;
        temp = one.x + two.y;
        return(temp);
}
void main()
{
        first a;
        second b;
        a.getdata();
        b.getdata();
        a.display();
        b.display();
        int te = sum(a,b);
        cout<<"Sum of the two
Private data variable (X + Y)";
        cout<<" = "<<te<<endl;

}
```

Inline Member Function

Inline functions are used in C++ to reduce the overhead of a normal function call.

A member function that is both declared and defined in the class member list is called an inline member function.

The inline specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.

The advantages of using inline member functions are:

1. The size of the object code is considerably reduced.
2. It increases the execution speed, and
3. The inline member function are compact function calls.

Inline Member Function

Syntax:

```
class user_defined_name
{
        private:
                ------------
        public:
inline return_type function_name(parameters);
inline retrun_type function_name(parameters);
                ---------------
                ---------------
};
```

Syntax

```
Inline return_type function_name(parameters)
{
        ----------------
        ----------------
}
```

Compiler may not perform inlining in such circumstances like:

1) If a function contains a loop. (for, while, do-while)

2) If a function contains static variables.

3) If a function is recursive.

4) If a function return type is other than void, and the return statement doesn’t exist in function body.

5) If a function contains switch or goto statement.

Example

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}


//Output: The cube of 3 is: 27
```

Inline function and classes

- It is also possible to define the inline function inside the class.

- All the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here.

- If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword

```
#include <iostream>
using namespace std;
class operation
{
   int a,b,add;

public:
   void get();
   void sum();
   };
inline void operation ::
get()
{
   cout << "Enter first
value:";
   cin >> a;
   cout << "Enter second
value:";
   cin >> b;
}
```

```
inline void operation :: sum()
{
   add = a+b;
   cout << "Addition of two numbers: " << a+b << "\n";
}

int main()
{
   cout << "Program using inline function\n";
   operation s;
   s.get();
   s.sum();
     return 0;
}
```

Output:

Enter first value: 45 Enter second value: 15 Addition of two numbers: 60 Difference of two numbers: 30 Product of two numbers: 675 Division of two numbers: 3

Virtual function

- Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform Late Binding on this function.
- Virtual Keyword is used to make a member function of the base class Virtual. Virtual functions allow the most specific version of a member function in an inheritance hierarchy to be selected for execution. Virtual functions make polymorphism possible.

Key:

- Only the Base class Method's declaration needs the Virtual Keyword, not the definition.
- If a function is declared as virtual in the base class, it will be virtual in all its derived classes.

01

Syntax

virtual return_type function_name (arg);

02

Example

```
virtual void show()
   {
      cout << "Base class\n";
   }
```

03

Note:

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

Virtual function features

Case 1:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual void display();
                virtual int sum();
}
virtual void sample::display()        //Error
{ }
```

Case 2:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual void display();
                virtual static int sum();      //error
}
int sample::sum()
{ }
```

Virtual function features

Case 3:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual sample(int x,float y);
        //error constructor
                void display();
                int sum();
}
```

Case 4:

```
class sample
{
        private:
                int x;
                float y;
        public:
                virtual ~sample(int x,float y);
        //invalid
                void display();
                int sum();
}
```

Virtual function features

Case 5:

```
class sample_1
{
        private:
                int x;
                float y;
        public:
                virtual int sum(int x,float y);
};
class sample_2:public sample_1
{
        private:
                int z;
        public:
                virtual float sum(int xx,float yy);
//error
};
```

Case 6:

```
virtual void display()        //Error, non member
function
{
        ----------------
        ----------------
}
```

Virtual Function Example

Example:

```
class Point
{
protected:
        float length,breath,side,radius,area,height;
};
class Shape: public Point
{
        public:
                virtual void getdata()=0;
                virtual void display()=0;
};
class Rectangle:public Shape
{
public:
        void getdata()
        {
                cout<<"Enter the Breadth Value:"<<endl;
                cin>>breath;
                cout<<"Enter the Length Value:"<<endl;
                cin>>length;
        }
        void display()
        {
                area = length * breath;
                cout<<"The Area of the Rectangle is:"<<area<<endl;
        }
};
```

```
class Square:public Shape
{
public:
void getdata()
{
        cout<<"Enter the Value of the Side of the Box:"<<endl;
        cin>>side;
}
void display()
{
        area = pow(side,4);
        cout<<"The Area of the Square is:"<<area<<endl;
        }
};
void main()
{
        Shape *s;
        Rectangle r;
        Square t;
        s = &r;
        s->getdata();
        s->display();
        s = &t;
        s->getdata();
        s->display();
}
```

Difference in invocation for virtual and non virtual function

Example:

```
class Base
{    public:
   void show()
   {
      cout << "Base class";
   }
};
class Derived:public Base
{    public:
   void show()
   {
      cout << "Derived Class";
   }
}
int main()
{
   Base* b;      //Base class pointer
   Derived d;    //Derived class object
   b = &d;
   b->show();    //Early Binding Occurs
}
```

```
class Base
{    public:
   virtual void show()
   {
      cout << "Base class\n";
   } };
class Derived:public Base
{    public:
   void show()
   {
      cout << "Derived Class";
   } };
int main()
{
   Base* b;      //Base class pointer
   Derived d;    //Derived class object
   b = &d;
   b->show();    //Late Binding Occurs
}
```

Difference in invocation for virtual and non virtual function

Example:

```
#include<iostream>
using namespace std;
class Base {
public:
  void foo()
  {
        std::cout << "Base::foo\n";
  }
  virtual void bar()
  {
        std::cout << "Base::bar\n";
  }
};

class Derived : public Base {
public:
  void foo()
  {
        std::cout << "Derived::foo\n";
  }
  virtual void bar()
  {
        std::cout << "Derived::bar\n";
  }
};

int main() {
   Derived d;
   Base* b = &d;
   b->foo(); // calls Base::foo
   b->bar(); // calls Derived::bar
}
```

Output:

```
Base::foo
Derived::bar
```

Override

Example:

```
#include <iostream>
using namespace std;

class Base {
public:

   // user wants to override this in the derived class
   virtual void func()
   {
      cout << "I am in base" << endl;
   }
};

class derived : public Base {
public:

   // did a mistake by putting   an argument "int a"
   void func(int a) override
   {
      cout << "I am in derived class" << endl;
   }
};

int main()
{
   Base b;
   derived d;
   cout << "Compiled successfully" << endl;
   return 0;
}
```

Output:

```
Base::foo
Derived::bar
```

Pure Virtual function

- Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function.
- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.
- Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

01

Syntax

```
virtual void f() = 0;
```

02

Example

```
class Base
{
   public:
   virtual void show() = 0;    // Pure
Virtual Function
};
```

Note:

03

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

Pure Virtual function

Example:

```
class pet
{
private:
        char name[5];
public:
virtual void getdata()=0;
virtual void display()=0;
};
class fish:public pet
{
private:
        char environment[10];
        char food[10];
public:
        void getdata();
        void display();
};
class dog: public pet
{
        private:
                char environment[10];
        char food[10];
        public:
                void getdata();
                void display();
};
void fish::getdata()
{
        cout<<"Enter the Fish Environment required"<<endl;
        cin>>environment;
        cout<<"Enter the Fish food require"<<endl;
        cin>>food;
}
void fish::display()
{
        cout<<"Fish Environment="<<environment<<endl;
        cout<<"Fish Food="<<food<<endl;
        cout<<"----------"<<endl;
}
```

Pure Virtual function

Example:

```
void dog::getdata()
{
        cout<<"Enter the Dog Environment required"<<endl;
        cin>>environment;
        cout<<"Enter the Dog Food require"<<endl;
        cin>>food;
}
void dog::display()
{
        cout<<"Dog Environment="<<environment<<endl;
        cout<<"Dog Food="<<food<<endl;
        cout<<"-----------------------------------"<<endl;
}
void main()
{
        pet *ptr;
        fish f;
        ptr=&f;
        ptr->getdata();
        ptr->display();
        dog d;
        ptr=&d;
        ptr->getdata();
        ptr->display();
        getch();
}
```

Pure Virtual function with definition

- Pure Virtual function is allowed to have definition in the base class itself, but the definition has to be made outside the class using scope resolution operator. Since inline pure virtual not supported compiler throws an error message.
- Pure virtual definition is useful when all the child will have certain common behavior to be implemented.

Example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
      virtual void display()=0;
};
void Base::display()
{
   cout<<"Base class version of pure virtual";
}

class Derived:public Base
{
        public:
                void display()
                {
                   cout<<"Derived Version of pure virtual function";
                }
};
int main()
{
        Derived ob;
        Base *p;
        p = &ob;
        p->display();
        return 0;
}
```

Pure Virtual function with definition

- Pure virtual implementation of base class version is called inside the override version of derived class method using class name and scope resolution operator

Example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
      virtual void display()=0;
};
void Base::display()
{
   cout<<"Base class version of pure virtual";
}

class Derived:public Base
{
        public:
        void display()
        {
                Base:: display();
              cout<<"Derived Version of pure virtual function";
        }
};
int main()
{
                Derived ob;
                Base *p;
                p = &ob;
                p->display();
                return 0;
}
```

Pure Virtual function with definition

Suppose that you're modelling a game system for an epic adventure, and your game has a variety of weapons (swords, arrows etc) that the hero uses to save the world from the evil.

You've decided to create an interface Weapon that models the abstract concept that must be the base for your weapons system. This interface offers the abstract member function attack and it needs to be completed by every concrete weapon in the game. The Weapon concept doesn't have a concrete meaning, but perhaps it's reasonable to have a default behavior for the attack that the concrete classes may use.

Example:

```
#include <iostream>

class Weapon {
        public:
                virtual ~Weapon() = default;
                virtual void attack() const = 0;
};

void Weapon::attack() const
{
        std::cout << "Default attack..\n";
}
```

```
class Sword : public Weapon
{
public:
        void attack() const override
        {
// Calls default member function (Weapon::attack)
                Weapon::attack();
                std::cout << "Sword attack...\n";
        }
};
```

Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure virtual function

Example:

```
/Abstract base class
class Base
{
   public:
   virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
   public:
   void show()
   {
      cout << "Implementation of Virtual Function in
Derived class\n";
   }
};
```

```
int main()
{
   Base obj;   //Compile Time Error
   Base *b;
   Derived d;
   b = &d;
   b->show();
}
```

State chart diagram

18CS202J OBJECT ORIENTED DESIGN AND PROGRAMMING

State diagram

- A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time. It's a **behavioural** diagram and it represents the behaviour using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams**.

Uses of state chart diagram

- State chart diagrams are useful to model reactive systems

  -Reactive systems can be defined as a system that responds to external or internal events.

- State chart diagram describes the flow of control from one state to another state.

Purpose

Following are the main purposes of using State chart diagrams:

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.

Difference between state diagram and flowchart

- The basic purpose of a **state diagram** is to portray various changes in state of the class and not the processes or commands causing the changes.

- However, a **flowchart** on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.

When to use State charts

- So the main usages can be described as:
- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.
- Forward and reverse engineering.

How to draw state charts

Before drawing a State chart diagram we must have clarified the following points:

- ✓Identify important objects to be analysed.
- ✓ Identify the states.
- ✓Identify the events.

Elements of state chart diagrams

- Initial State: This shows the starting point of the state chart diagram that is where the activity starts.

Elements of state chart diagrams

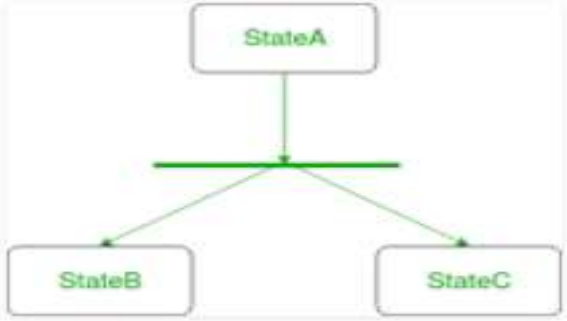
- State: A state represents a condition of a modelled entity for which some action is performed. The state is indicated by using a rectangle with rounded corners and contains compartments

Elements of state chart diagrams

- **Composite state –** We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.

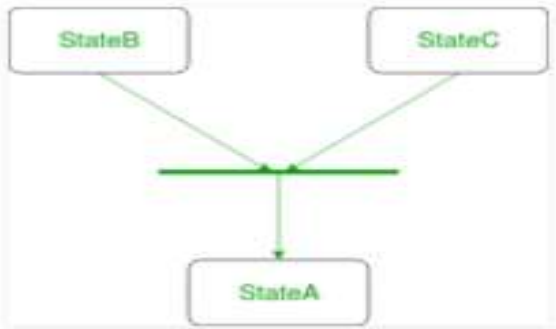
Elements of state chart diagrams

- **Fork –** We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



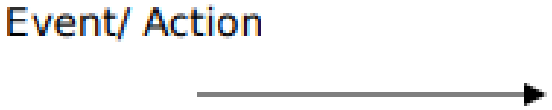
Elements of state chart diagrams

- **Join –** We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.



Elements of state chart diagrams

- Transition: It is indicated by an arrow. Transition is a relationship between two states which indicates that Event/ Action an object in the first state will enter the second state and performs certain specified actions.

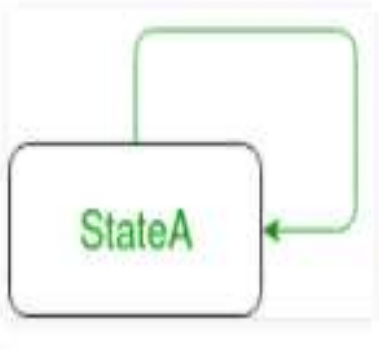


Elements of state chart diagrams

- **Transition –** We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.

Elements of state chart diagrams

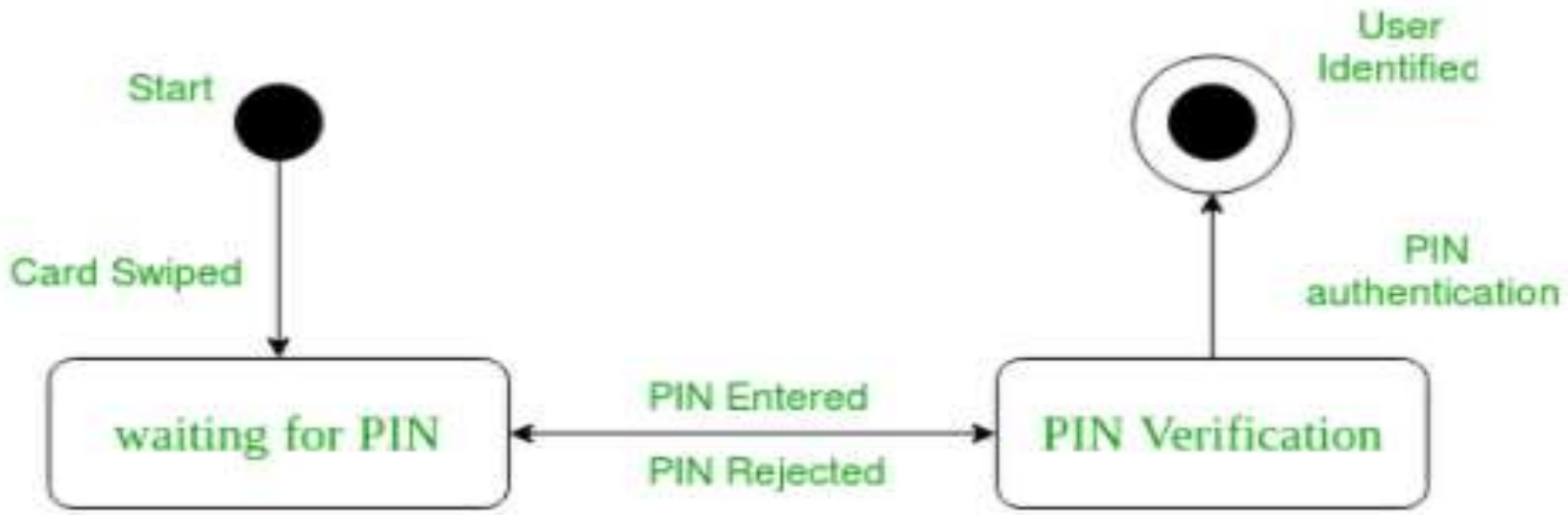
- **Self transition –** We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.



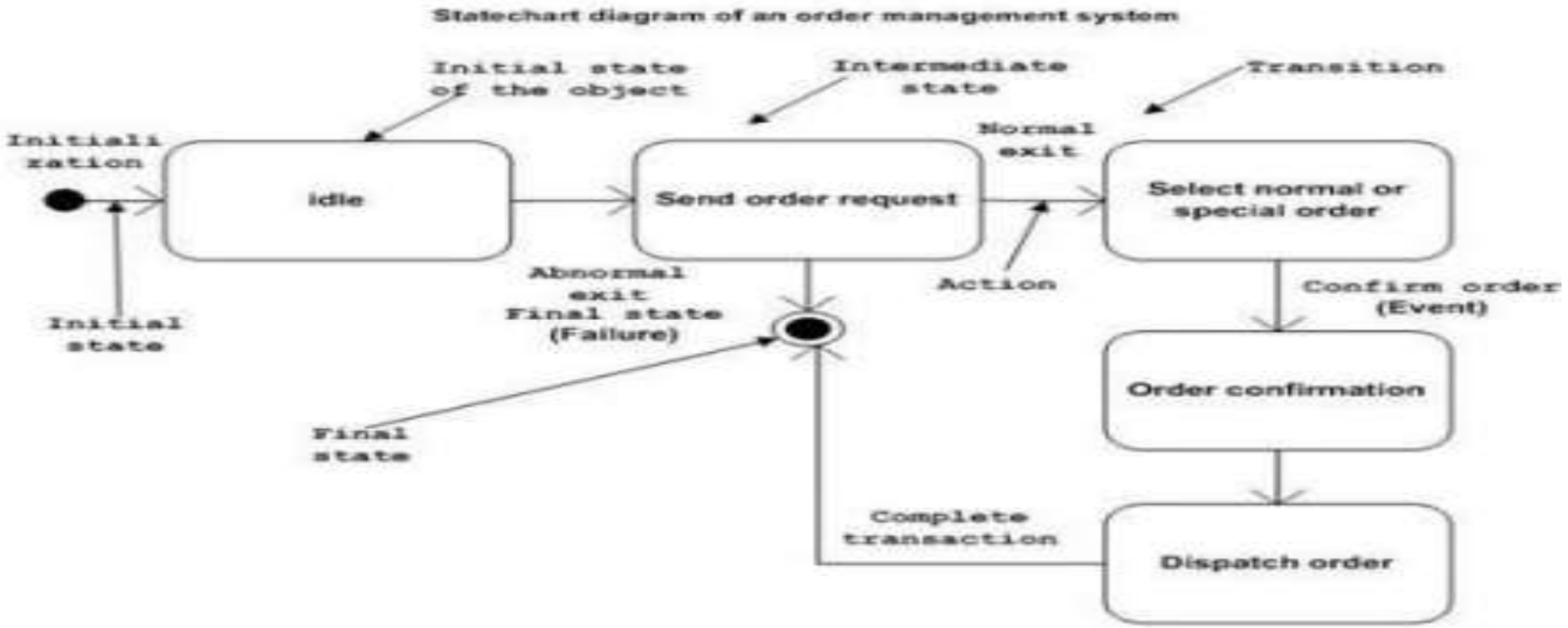
Elements of state chart diagrams

- Final State: The end of the state chart diagram is represented by a solid circle surrounded by a circle.

Example state chart for ATM card PIN Verification



Example state chart for order management system



ACTIVITY DIAGRAM

18CSC202

Object Oriented Design and Programming

Activity Diagram

- Activity diagram is UML behavior diagram which emphasis on the sequence and conditions of the flow
- It shows a sequence of actions or flow of control in a system.
- It is like to a flowchart or a flow diagram.
- It is frequently used in business process modeling. They can also describe the steps in a use case diagram.
- The modeled Activities are either sequential or concurrent.

Benefits

- It illustrates the logic of an algorithm.
- It describes the functions performed in use cases.
- Illustrate a business process or workflow between users and the system.
- It Simplifies and improves any process by descriptive complex use cases.
- Model software architecture elements, such as method, function, and operation.

Symbols and Notations

Activity

- Is used to illustrate a set of actions.
- It shows the non-interruptible action of objects.

Activity

Symbols and Notations

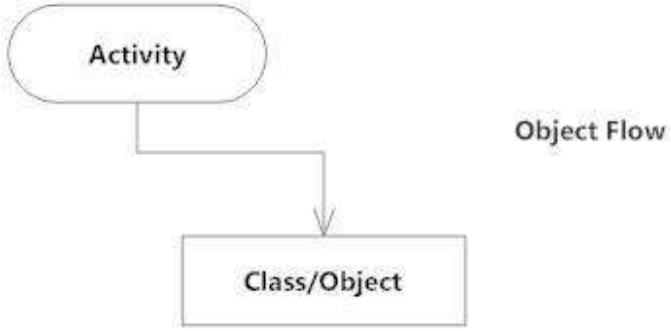
Action Flow

- It is also called edges and paths
- It shows switching from one action state to another. It is represented as an arrowed line.

Symbols and Notations

Object Flow

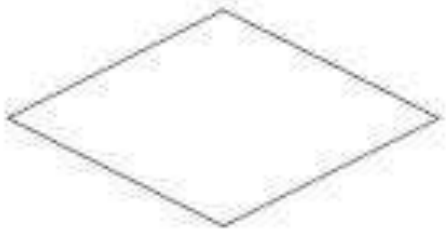
- Object flow denotes the making and modification of objects by activities.
- An object flow arrow from an action to an object means that the action creates or influences the object.
- An object flow arrow from an object to an action indicates that the action state uses the object.



Symbols and Notations

Decisions and Branching

- A diamond represents a decision with alternate paths.
- When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.
- The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."

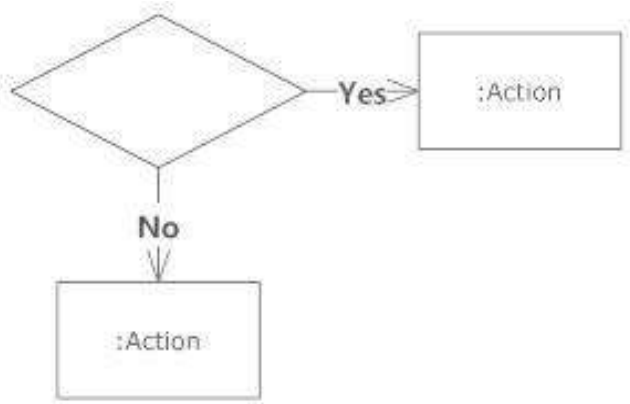


Decision Symbol

Symbols and Notations

Guards

- In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity.
- These are not essential, but are useful when a specific answer, such as "Yes, three labels are printed," is needed before moving forward.



Guard Symbols

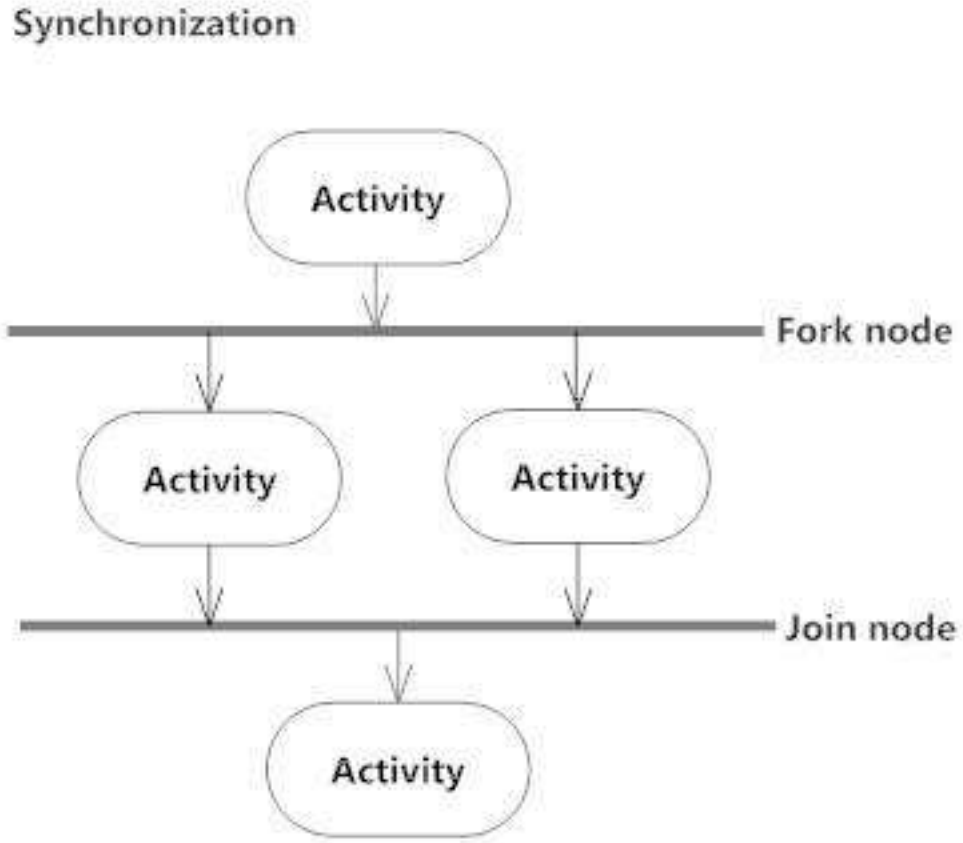
Symbols and Notations

Synchronization

- A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A join node joins multiple concurrent flows back into a single outgoing flow.
- A fork and join mode used together are often referred to as synchronization.

Symbols and Notations

Synchronization



Symbols and Notations

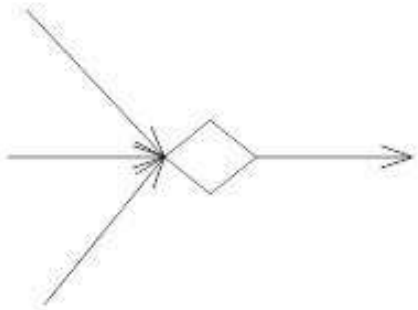
Time Event

- This refers to an event that stops the flow for a time; an hourglass depicts it.

Symbols and Notations

Merge Event

- A merge event brings together multiple flows that are not concurrent.



Merge

Final State or End Poi

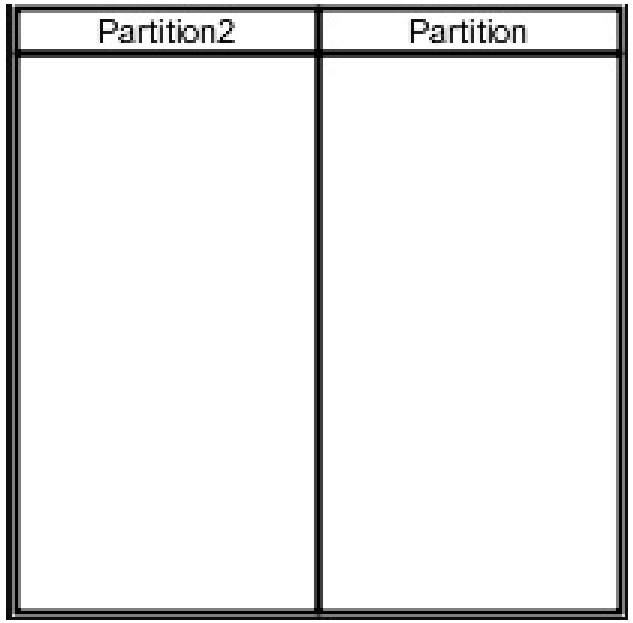
- An arrow pointing to a filled circle nested inside another circle represents the final action state.

End Point Symbol

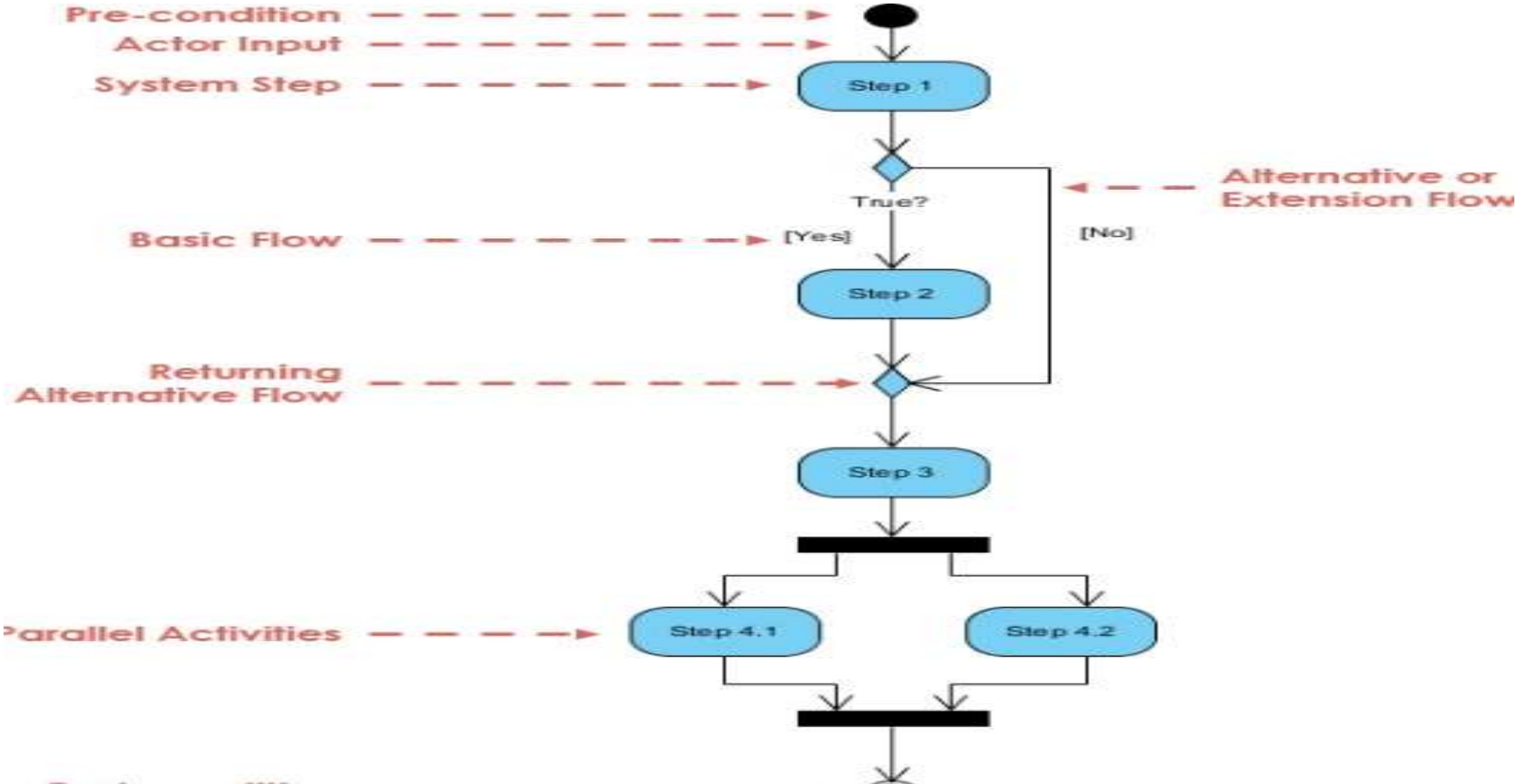
Symbols and Notations

Swimlane and Partition

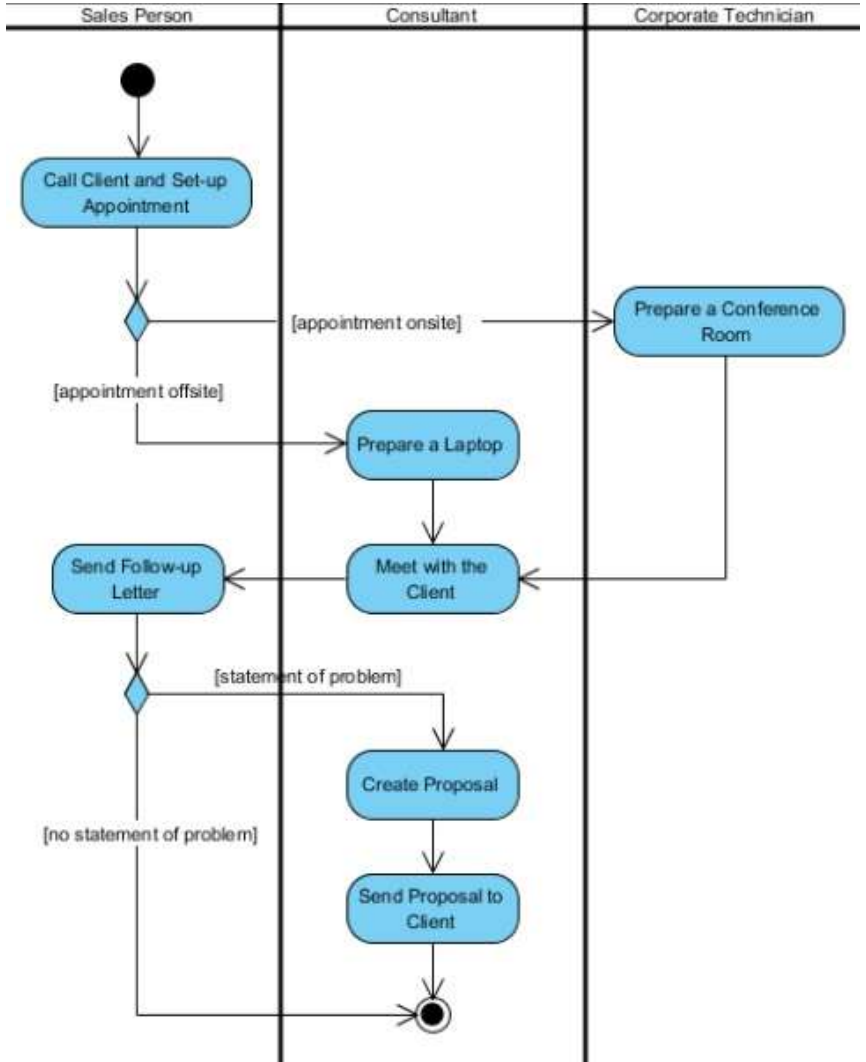
- A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread



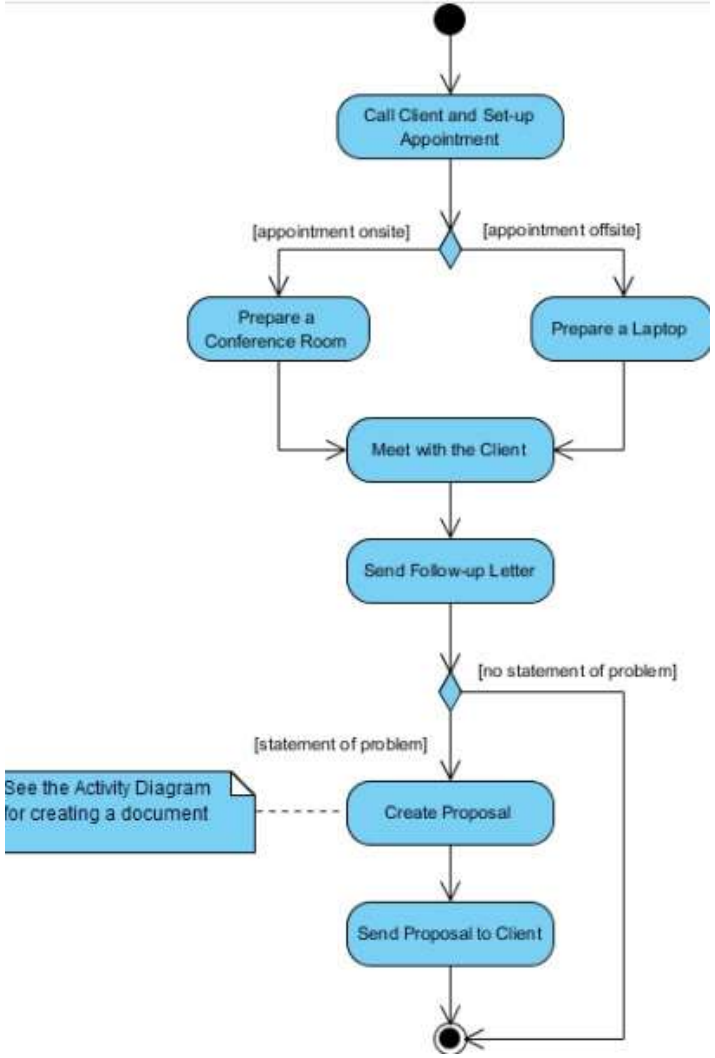
Activity Diagram



Activity Diagram with Swimlane



Activity Diagram without Swimlane



State Chart and Activity Diagram Scenarios

6.4 PROBLEM STATEMENT: MCA ADMISSION SYSTEM

MCA admission procedure as controlled by Directorate of Technical Education (DTE) is as follows:

1. DTE advertises the date of MCA entrance examination.
2. Student has to apply for the entrance examination.
3. Results are declared by DTE.
4. Student has to fill up the option form to select the college of his/her choice.
5. DTE displays the allotment list in the web site and intimation to all colleges.
6. Students should report the allotted colleges and complete the admission procedure.

1. Analysis of MCA Admission System

Drawing an activity diagram for the whole system we,

1. Find out swimlanes if any. To find swimlanes, see if we can span some activities over different organizational units/places.
2. Find out in which swimlane the admission process begins and where it ends. Those will become the initial and final states.
3. Then, identify activities occurring in each swimlane. Arrange activities in sequence flow spanning over all the swimlanes.
4. Identify conditional flow or parallel flow of activities. Parallel flow of activities must converge at a single point using join bar.
5. During the activities are performed, if any document is generated or used, take it as an object and show the object flow.

Swimlanes identified for admission process are shown in Figure 6.9.

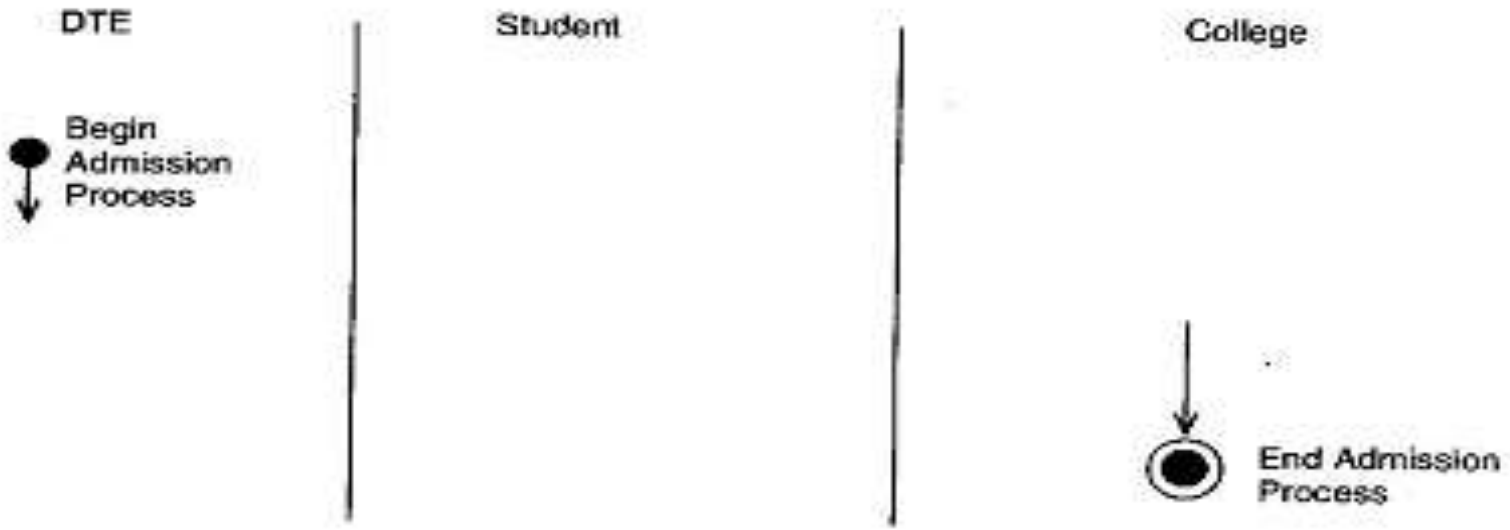


FIGURE 6.9 Swimlanes in admission process.

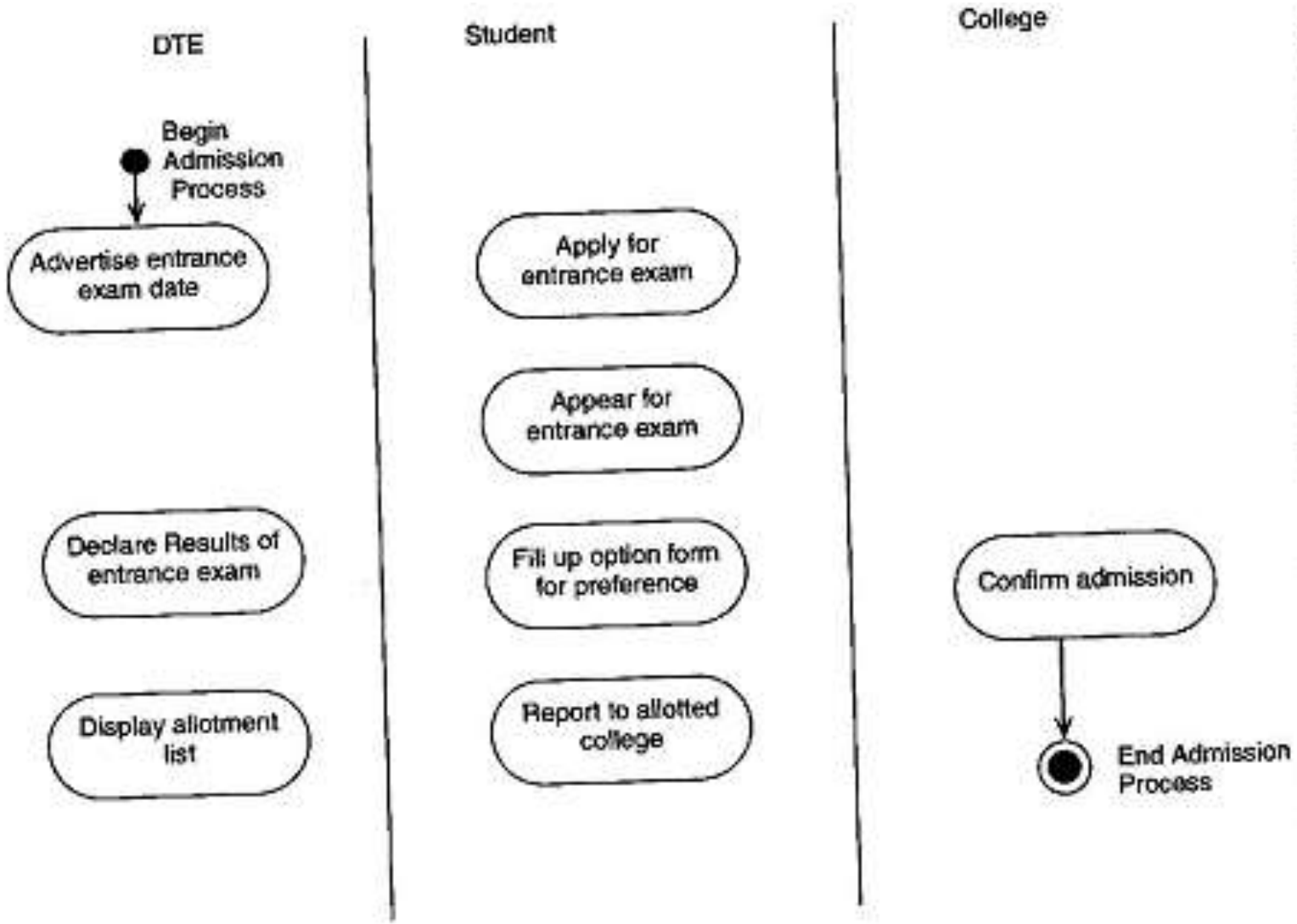


FIGURE 6.10 Activities in admission process.

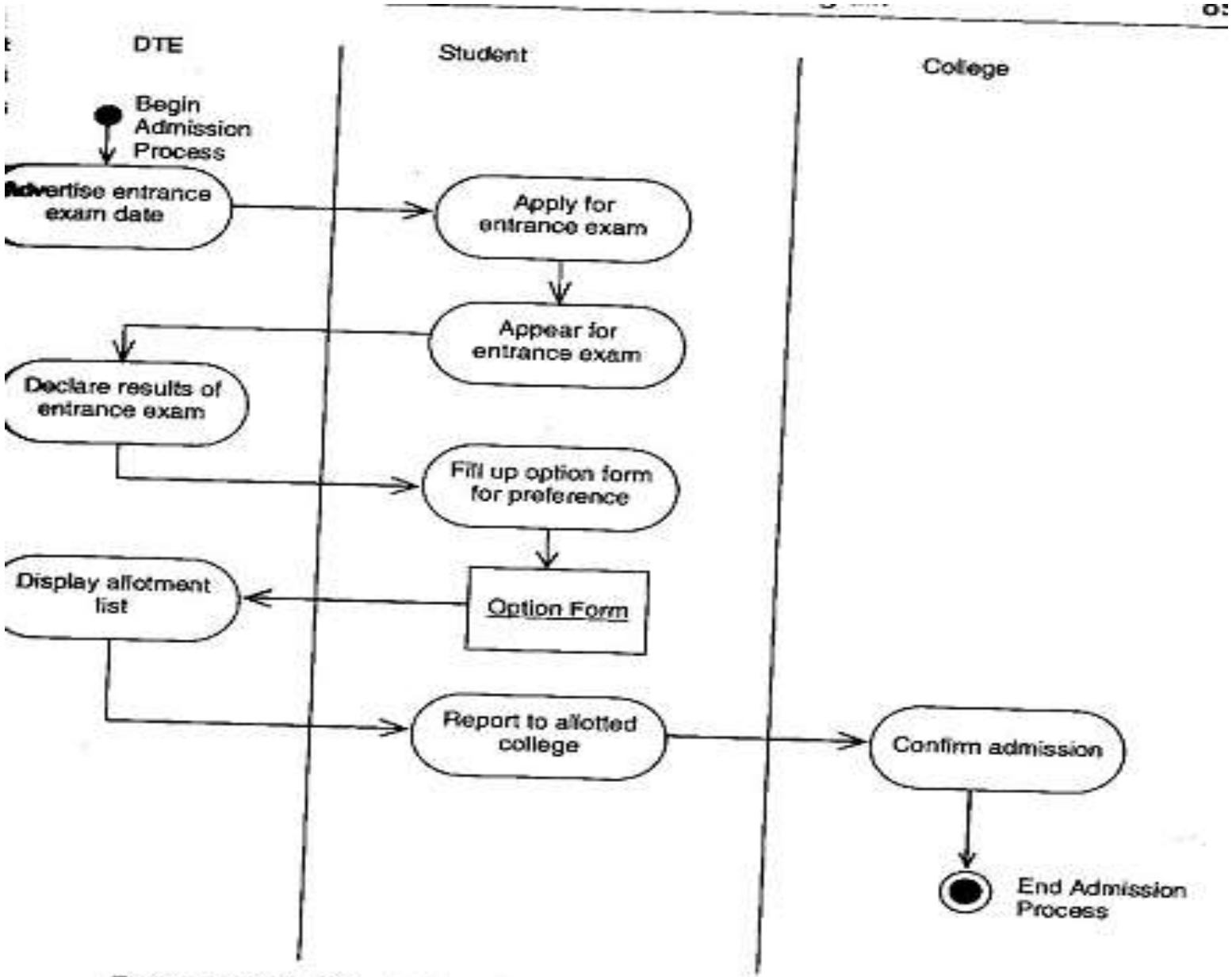


FIGURE 6.12 Complete activity diagram of admission process.

State Chart Diagram

For example, a simple state diagram for a Door object with states *Opened*, *Closed* and *Locked* is shown in Figure 6.15(b). All the three states of the door are simple states without any substates.

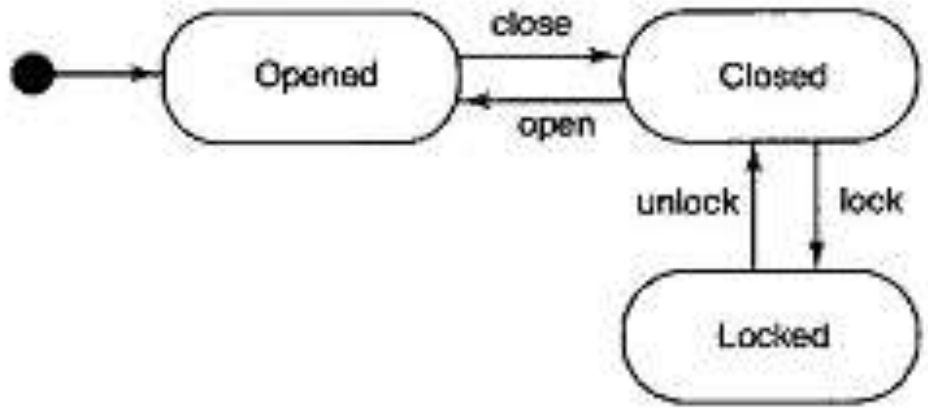


FIGURE 6.15(b) State diagram for a door showing simple states.

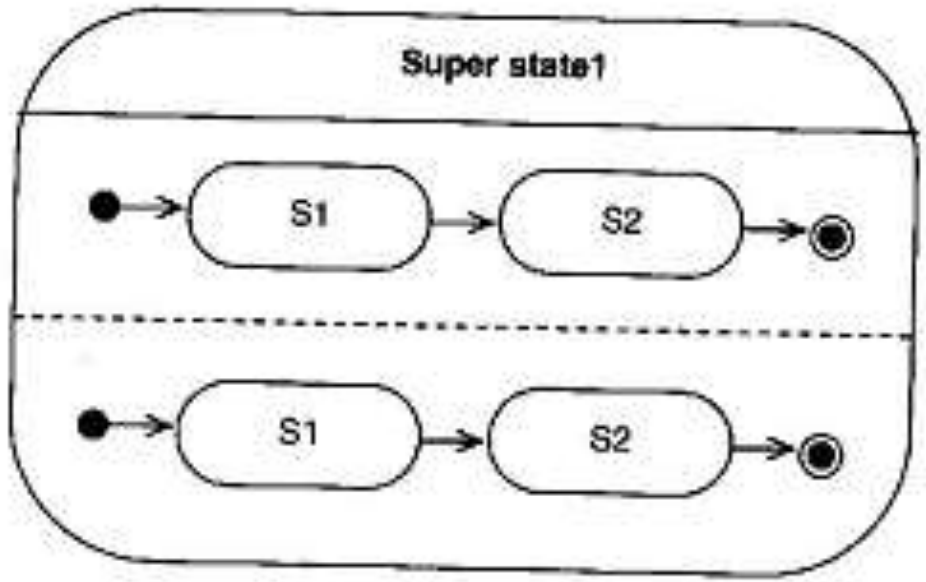


FIGURE 6.16(a) Representing super state concurrent composite state.

For example, Figure 6.16(b) shows a statechart diagram for a gas station where on arrival for filling gas, attendants can perform two tasks in parallel, filling gas as well as washing windshield. InService is the concurrent composite state having Filling Gas Tank and Washing Windshield substates in parallel.

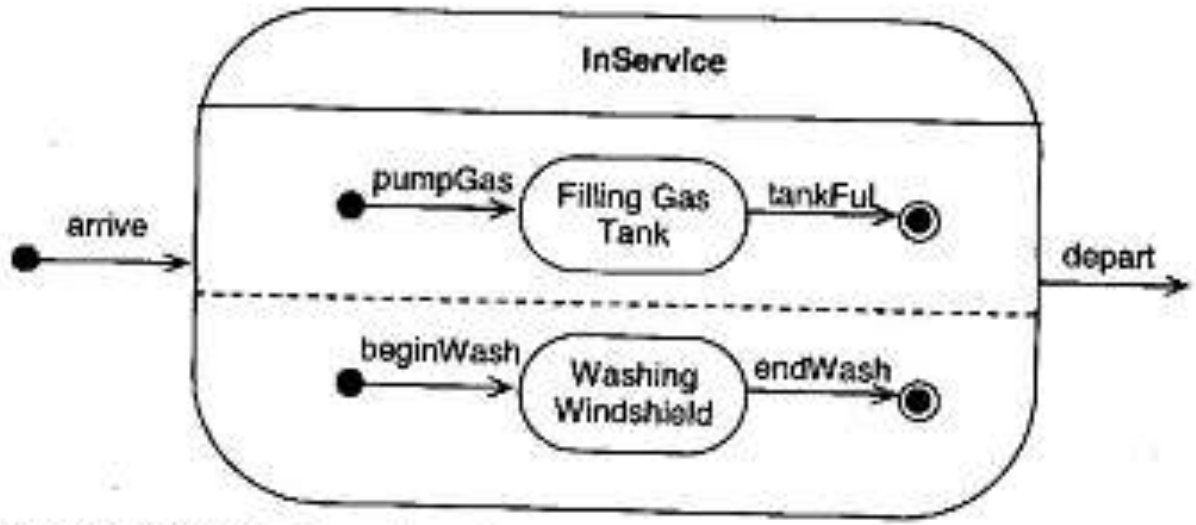


FIGURE 6.16(b) Example of concurrent composite state—InService.

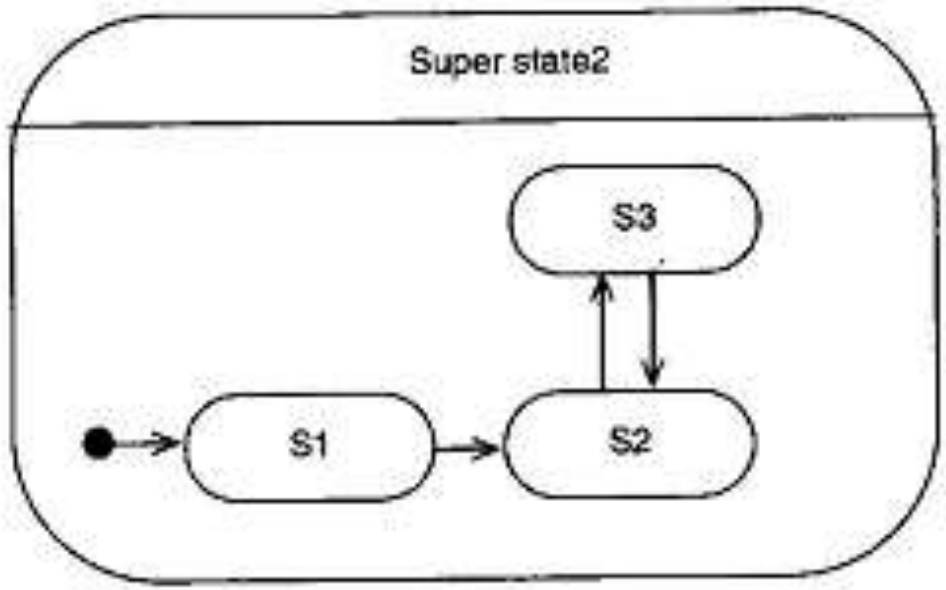


FIGURE 6.16(c) Representing superstate—sequential composite state.

6.8 PROBLEM STATEMENT: ADVERTISEMENT CAMPAIGN OF ABC PVT. LTD.

ABC Pvt. Ltd. company wants to start its advertisement campaign. The advertisement will be prepared. After the approval of the advertisement, the advertisement will be scheduled for publication. The advertisement will be published after scheduling is done.

6.8.1 Analysis of Advertisement Campaign of ABC Pvt. Ltd.

In this problem statement, the object for which a statechart diagram should be drawn is:

Advertisement campaign

Since the *Advertisement campaign* has fixed states from start till end, the statechart diagram will be of type one shot life cycle statechart diagram. Hence there will be one initial state and one or more final state.

Initial state: Figure 6.18(a) shows the initial state of the advertisement compain process.

FIGURE 6.18(a) Begin advertisement campaign process.

Final state: Figure 6.18(b) shows the final state of the compain process.

FIGURE 6.18(b) End advertisement campaign process.

Intermediate states: Figure 6.19(a) shows the intermediate states of the process.

FIGURE 6.19(a) Intermediate states—advertisement campaign.

Figure 6.19(b) shows all the transitions of the advertisement compain process.

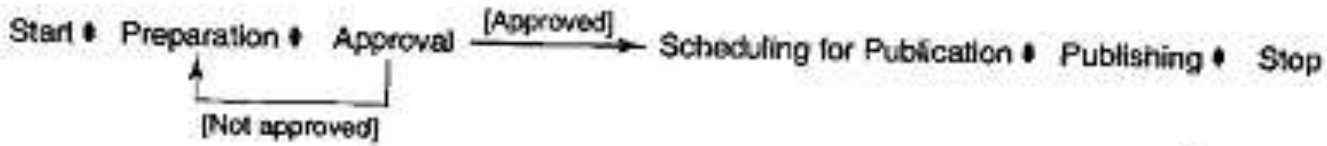


FIGURE 6.19(b) Transition—advertisement campaign: scheduling for publication and publishing.

The complete state transition diagram for the *advertisement campaign* object is shown in Figure 6.20.

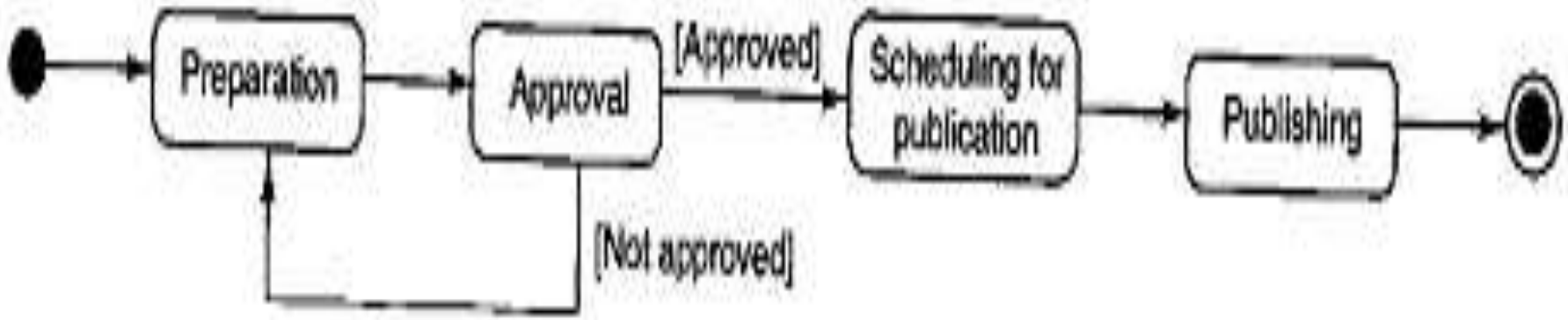


FIGURE 6.20 State chart diagram—advertisement campaign.