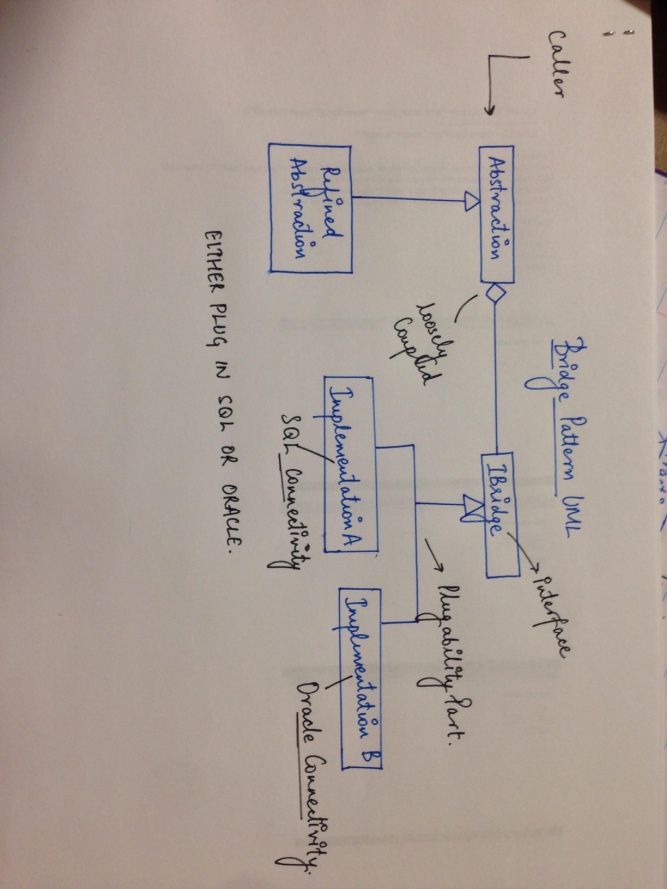
**BRIDGE PATTERN**

The purpose is to decouple an abstraction from its implementation so that the two can vary

Independently. In the Bridge pattern, we separate an abstraction and its implementation and develop separate inheritance structures for both the abstraction and the implementer.

Whenever there is plug ability or injection or loose coupling is involved, Aggregation will be involved. In Bridge Pattern, Caller calls the Abstraction which therefore, is aggregated or loosely coupled with Refined Abstraction and the interface say IBridge. Also there are involved two or more than two implementations say A, B or C and so on.

IBridge now, can be implemented by either Implementation A or Implementation B or Implementation C as shown in the picture given below.



CODE:

Class DataAbstraction

{

**// we will not create object here but will inject the object through constructor**

**}**

Public interface IBridgeServer

{

// code for database queries

//two methods

Public int GetSingleReturn(string sql);

Public object GetManyReturns(DataTable dt);

}

Class MyDataBase : IBridgeServer

{

DataAbstraction da = new DataAbstraction( new SQL Connection);

// OR

DataAbstraction da = new DataAbstraction( new ORACLE Connection);

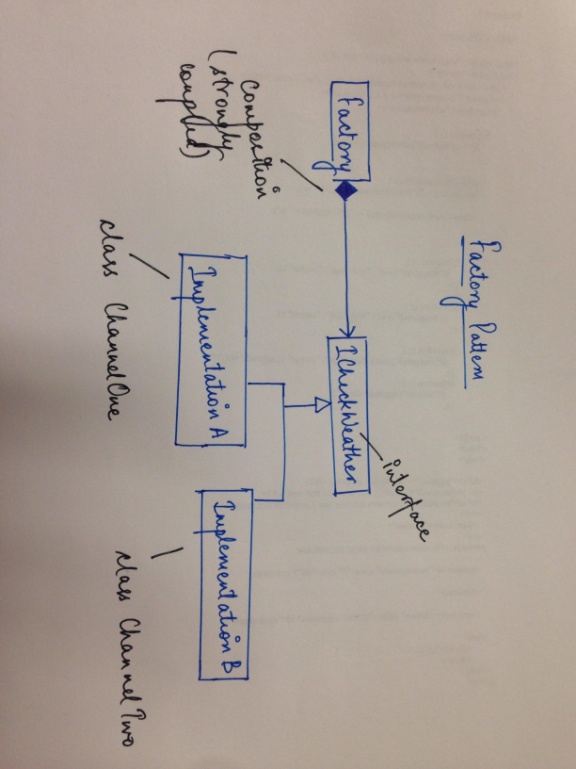
// Code to connect to DATABASE.

}

**FACTORY PATTERN**

The job of the Factory Design Pattern is to create the object of the class on behalf of the caller. Caller will specify the request and factory will generate object of the class which will meet the requirement.

In the picture given below, class ChannelOne would implement the interface ICheckWeather and also does the ChannelTwo class.

****

**CODE:**

Interface ICheckWeather

{

Int GetMaxTemp(string zipcode); }

Class ChannelOne : ICheckWeather

{

Public int GetMaxTemp(string zipcode)

{

Int maxTemp=0;

If(zipcode == “06604”)

maxTemp = 52;

if (zipcode == “06601”)

{

// so on

// many if conditions

}

Return MaxTemp;

}

Class ChannelTwo : ICheckWeather

{

// Same Code as above

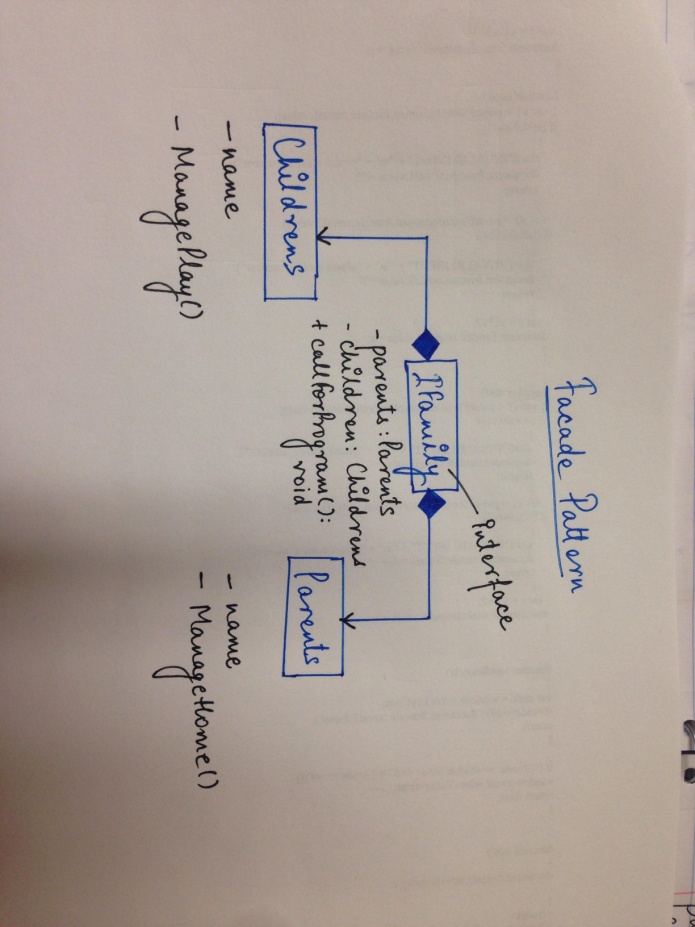
}

**FAÇADE DESIGN PATTERN** (optional)

Façade Pattern is used in two situations:

1. When there is a complex system, which consist of many sub-systems. Façade therefore provides a simplified interface to the system.
2. To provide safe access to the resource.

A design pattern is helpful to solve the problem that occurs in software design. It gives the direction to solve the recurring problems in a methodical way. In object oriented software, the object should be discovered to design the software which is the difficult task.



The class Family is the Façade that structures the Children and the Parents classes as part of the Family. The clients simply communicate with the family.

CODE:

#include<iostream>

#include<string>

Using namespace std;

class Parents

{

void ManageHouseHold(){

cout<<”management of household thing”<<endl;

}

public:

void talk(){

cout<<”talk about the progress of children with parents”<<endl;

}

};

class Childrens

{

String name;

public:

void classes(){

cout<<”the classes their children study”<<endl;

}

void grades(){

cout<<”the grades of the children”<<endl;

}

void preparation(){

cout<<”dance/song/decoration/play”<<endl;

}

};

Class Relation

{

public:

void uncle(){

cout<<”He is your uncle”<<endl;

}

void aunt(){

cout<<”She is your aunt”<<endl;

}

void siblings(){

cout<<”They are your siblings”<<endl;

}

};

//Façade Class

class Family

{

Parents parent;

Children children;

Relation relation ;

public:

void CallForProgram()

{

public:

parents.manageHouseHold();

children.preparation();

}

};

int main()

{

Family family;

family.callforprogram();

return 0;

}