**Open Closed Principle**

The open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

Open/closed principle has been used in two ways.

1. Meyer's open/closed principle
2. Polymorphic open/closed principle

Both ways use generalizations (for instance, inheritance or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.

1. **Meyer's open/closed principle**

A module will be said to be **open if it is still available for extension**. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.

A module will be said to be **closed if it is available for use by other modules**. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.

1. **Polymorphic open/closed principle**

In contrast to Meyer's usage, this definition advocates inheritance from abstract base classes.

Interface specifications can be reused through inheritance but implementation need not be. The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface.

**Why OCP?**

If not followed:

1. End up testing the entire functionality
2. QA team need to test the entire flow
3. Costly process for the organization
4. Breaks the single responsibility as well
5. Maintenance overheads increases on the classes

# Example

A class is designed to calculate bonus of an employee

class Employee {

int id;

string name;

Employee() {};

Employee(int ID, string Name) : id(ID), name(Name) {}

double calculate\_bonus(double salary) {return (salary \* 0.1)}

};

Now we have a requirement to calculate bonus of permanent and contract employee, so we need to modify this class and add an element employee type and calculate bonus based on the employee type

class Employee {

int id;

string name;

string employee\_type;

Employee() {};

Employee(int ID, string Name, string etype) : id(ID), name(Name), employee\_type(etype) {}

double calculate\_bonus(double salary) {

if("Permanent" == employee\_type) return (salary \* 0.1);

else return (salary \* 0.05);

}

};

Although this code work but if getv more requirement and enhancing the same class we end up facing the problems we have discussed above.

**How to fulfill new requirements?**

Make the employee class as abstract base class and implement new employee classes based on the diffrenet bonus

class Employee {

int id;

string name;

Employee() {};

Employee(int ID, string Name) : id(ID), name(Name) {}

double calculate\_bonus(double salary) = 0;

};

class PermanentEmployee : public Employee {

PermanentEmployee() {}

PermanentEmployee(int ID, string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) {

return (salary \* 0.1);

}

};

class TemporaryEmployee : public Employee {

TemporaryEmployee() {}

TemporaryEmployee(int ID, string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) {

return (salary \* 0.05);

}

};

# END