Dependency Inversion Principle

High level Modules should not depend on low level modules. Both should depend on abstraction. Abstraction should not depend the details. Details should depend on the abstraction.

Let's consider an example of a Database class that stores user information. We could define a concrete implementation of this class that depends on the MySQLConnector class.

```
Class MySQLConnector
{
   public:
      void connect(){
      }
      void query(const string& query)
       {
        }
};

class Database
{
   public:
       void addUser(const string& name,const string& email)
        {
            MySQLConnector connector;
             connector.connect();
             string query = "INSERT INTO user(name,email)VALUES('"+name+"'"+email'");
             connect.query(query);
         }
};
```

This implementation violates the Dependency Inversion Principle, as the Database class depends on the low-level MySQLConnector class. If we were to switch to a different database system, we would need to modify the Database class to use a different connector.

To adhere to the DEPENDENCY INVERSION PRINCIPLE we could define an abstract interface for the database connector, and have Database class depend on this interface instead of concrete implementation.

```
class IDatabaseConnector
{
    public:
        virtual void connect()=0;
         virtual void query(const string& query)=0;
};

class MySQLConnector : IdatabaseConnector
{
    public:
        void connect() {
            cout<<"Connecting.mysql"<<endl;
         }
         void query(const string& query) {
             cout<<"query.execute()"<<endl;
         }
 };

class Database
{
   private:
       IDatabaseConnector& m_connect;
   public:
       Database(IDatabaseConnector& connect): m_connect(connect){}
       void addUser(const string& name,const string& email)
        {
               m_connect.connect();
               string query=INSERT INTO user(name,email)VALUES('"+name+'"email'");
               m_connect.query(query);
        }
};
```

Now, the Database class depends on the abstract IDatabaseConnector interface instead of a concrete implementation. We can create different implementations of the IDatabaseConnector interface for different database systems, and easily swap them out without modifying the Database class

```
class PostgreSQL : public IdatabaseConnector
{
   public:
      void connect() {
           cout<<"Connecting.mysql"<<endl;
         }
         void query(const string& query) {
             cout<<"query.execute()"<<endl;
         }
};
```

```
int main()
{
    PostgreSQL m_postgreSQL;
    Database db(m_postgreSQL);

     db.addUser(“John doe”,123@gmail.com”);
}
```

By adhering to the Dependency Inversion Principle, we can create more flexible and maintainable code that is easier to extend and modify. High-level modules depend on abstractions, not low-level details, which allows for easier substitution of implementations.