软件构造实验报告一

实验名称：抽象工厂模式与单件模式编程实现

实验时间：2019.4.3

学号：E21614061

姓名：徐奕

所在院系：计算机科学与技术学院

所在专业：软件工程

【实验目的和要求】

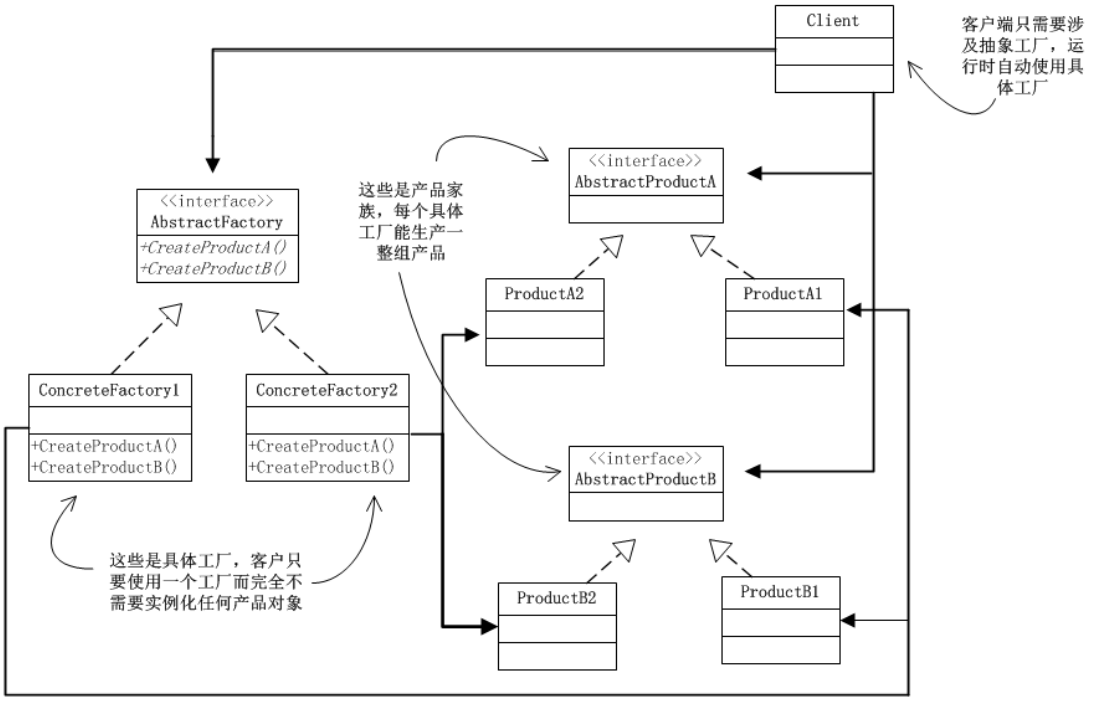
a）熟悉并理解抽象工厂模式的原理与动机

b）用高级语言实现抽象工厂模式

【实验原理】

- 抽象工厂

当每个抽象产品都有多于一个的具体子类的时候，工厂角色怎么知道实例化哪一个子类呢？比如每个抽象产品角色都有两个具体产品。抽象工厂模式提供两个具体工厂角色，分别对应于这两个具体产品角色，每一个具体工厂角色只负责某一个产品角色的实例化。每一个具体工厂类只负责创建抽象产品的某一个具体子类的实例。



- 单件模式

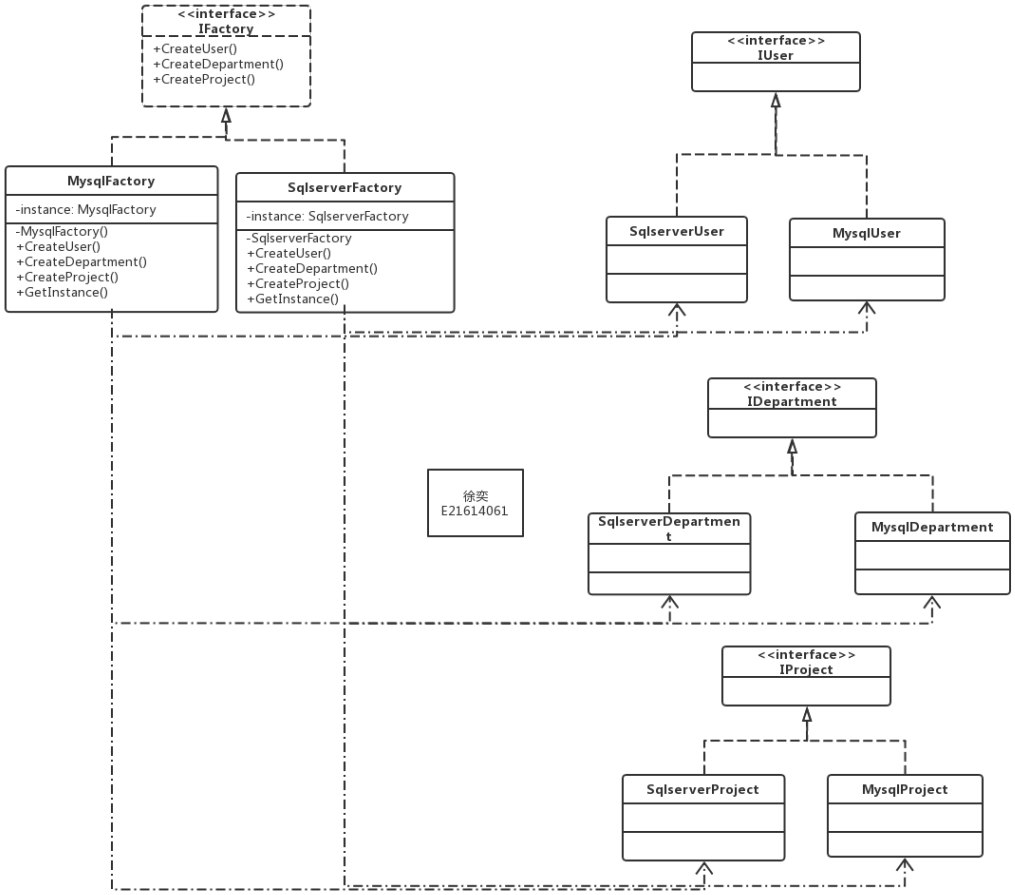
Singleton 就是确保一个类只有唯一的一个实例。Singleton 主要用于对象的创建，这意味着，如果某个类采用了 Singleton 模式，则在这个类被创建后，它将有且仅有一个实例可供访问。很多时候我们都会需要 Singleton 模式，最常见的比如我们希望整个应用程序中只有一个连接数据库的 Connection 实例；又比如要求一个应用程序中只存在某个用户数据结构的唯一实例。我们都可以通过应用 Singleton 模式达到目的。

【实验内容】

该公司数据库拥有三张表，分别是用户表、部门表和项目表。每张表的操作都支持查询和添加功能。数据库支持 MySQL 和 SQL Server 两种。结合抽象工厂模式和单件模式给出该系统的模拟代码。

在抽象工厂模式中，一个应用里一般每个产品只需要一个具体工厂的实例，因此，工厂通常最好用单件模式实现。实验要求结合抽象工厂模式和单件模式，模拟公司数据库创建过程。

【实验 UML 图】



【实验代码与函数】

① IFactory 抽象工厂

```
class IFactory{
       public:
           virtual ~IFactory(){cout<<“抽象工厂实例删除！”<<endl;}
           virtual IUser* CreateUser()=0;
           virtual IDepartment* CreateDepartment()=0;
           virtual IProject* CreateProject()=0;
};
```

② 具体与抽象的项目类

```
       class User{
           int id;
       };
```

```
class Department{
    int id;
};

class Project{
    int id;
};




class IUser{
public:
    virtual void Insert(User* user) = 0;
    virtual User* GetUser(int id) = 0;
};

class IDepartment{
public:
    virtual void Insert(Department* department) = 0;
    virtual Department* GetDepartment(int id) = 0;
};

class IProject{
public:
    virtual void Insert(Project* project) = 0;
    virtual Project* GetProject(int id) = 0;
};
```

③ SqlServer 与 Mysql 生成具体项目

```
class SqlserverUser: public IUser{
public:
    void Insert(User* user){
        cout<<“增加一个用户”<<endl;
    }
    User* GetUser(int id){
        cout<<“得到一个用户”<<endl;
        return NULL;
    }
};
```

```
class SqlserverDepartment: public IDepartment{
public:
    void Insert(Department* department){
        cout<<“增加一个部门”<<endl;
    }
    Department* GetDepartment(int id){
        cout<<“得到一个部门”<<endl;
        return NULL;
    }
};



class SqlserverProject: public IProject{
public:
    void Insert(Project* project){
        cout<<“增加一个项目”<<endl;
    }
    Project* GetProject(int id){
        cout<<“得到一个项目”<<endl;
        return NULL;
    }
};



class MysqlUser: public IUser{
public:
    void Insert(User* user){
        cout<<“增加一个用户”<<endl;
    }
    User* GetUser(int id){
        cout<<“得到一个用户”<<endl;
        return NULL;
    }
};



class MysqlDepartment: public IDepartment{
public:
    void Insert(Department* department){
```

```
            cout<<“增加一个部门”<<endl;
        }
        Department* GetDepartment(int id){
            cout<<“得到一个部门”<<endl;
            return NULL;
        }
    };



    class MysqlProject: public IProject{
    public:
        void Insert(Project* project){
            cout<<“增加一个项目”<<endl;
        }
        Project* GetProject(int id){
            cout<<“得到一个项目”<<endl;
            return NULL;
        }
    };
```

④ 具体工厂类

```
    class SqlserverFactory: public IFactory{
    public:
        ~SqlserverFactory(){cout<<“SQL Server 抽象工厂实例删
除！”<<endl;}
        IUser* CreateUser(){
            return new SqlserverUser();
        }
        IDepartment* CreateDepartment(){
            return new SqlserverDepartment();
        }
        IProject* CreateProject(){
            return new SqlserverProject();
        }
        static IFactory *GetInstance(){
            if (factory == NULL ){
                    return new SqlserverFactory();
                    cout<<“抽象工厂实例生成！”<<endl;

            }else cout<<“Error, 抽象工厂实例已有！”<<endl;
            return factory;
        }
    private:
```

```
    SqlserverFactory(){}
    static SqlserverFactory* factory;
};
SqlserverFactory* SqlserverFactory::factory = NULL;
```

```
class MysqlFactory: public IFactory{
public:
    ~MysqlFactory(){cout<<“SQL  Server  抽象工厂实例删除！”<<endl;}
    IUser* CreateUser(){
        return new MysqlUser();
    }
    IDepartment* CreateDepartment(){
        return new MysqlDepartment();
    }
    IProject* CreateProject(){
        return new MysqlProject();
    }
    static IFactory *GetInstance(){
        if (factory == NULL ){
            cout<<“抽象工厂实例生成！”<<endl;
            factory =  new MysqlFactory();

        }else cout<<“Error, 抽象工厂实例已有！”<<endl;
        return factory;
    }
private:
    MysqlFactory(){}
    static MysqlFactory* factory;
};
MysqlFactory* MysqlFactory::factory = NULL;
```

⑤ 客户端（仅以 MySQL 为例）

```
int main(){
    User* user = new User();
    Department* dept = new Department();
    Project* pro = new Project();
```

```
        cout<<endl<<“MySQL”<<endl;
        IFactory *factory1 = MysqlFactory::GetInstance();
        IFactory *factory2 = MysqlFactory::GetInstance();
        //生成第二个失败
        IUser* iu = factory1->CreateUser();
        iu->Insert(user);
        iu->GetUser(1);
        IDepartment* idept = factory1->CreateDepartment();
        idept->Insert(dept);
        idept->GetDepartment(1);

        IProject* ipro = factory1->CreateProject();
        ipro->Insert(pro);
        ipro->GetProject(1);
        return 0;
}
```

【实验结果】

注意第二行是再次生成一个 MySQL 的工厂，但是由于只能有一个，因此不能再次创建。

```
MySQL
抽象工厂实例生成!
Error, 抽象工厂实例已有!
增加一个用户
得到一个用户
增加一个部门
得到一个部门
增加一个项目
得到一个项目

SQL Server
Error, 抽象工厂实例已有!
Error, 抽象工厂实例已有!
增加一个用户
得到一个用户
增加一个部门
得到一个部门
增加一个项目
得到一个项目
```

【实验总结】

① 注意使用单件模式写 MySQL 和 SQL Server 的具体工厂时，只能对自身单件，即生成一个 MySQL 工厂时，**不能再次生成第二个 MySQL 工厂，但是可以生成 SQL Server 工厂。**

② 当使用单件模式时，具体工厂的构造函数需要放在 private 中，这样客户端不能随意创建多个工厂。在 public 中需要增加 getInstance 函数用来判断是否生成了工厂，如果存在，则不能创建第二个，注意这是 static 类型。

③ 具体工厂的指针需要在类外部全局部分进行初始化为 NULL。