

软件架构与设计模式

软件工程系 刘慰

liuwei@nbu.edu.cn

工厂方法模式可以继续改进吗？

- 新的需求

在“工厂方法模式”的课程中，有一个不同类型动力小汽车的例子。如果：

汽车上除了安装发动机以外，还需要安装发电机（**Dynamo**），而发电机同样分为

1. 汽油发电机（**GasolineDynamo**）
2. 太阳能发电机（**SolarDynamo**）
3. 核能发电机（**NuclearDynamo**）

那么程序应该如何修改？

工厂方法模式可以继续改进吗？

Benz 和 BMW 都生产三厢车（Sedan）、两厢车（Hatchback）和越野车（SUV）这三种类型的小汽车（Car）。

Benz

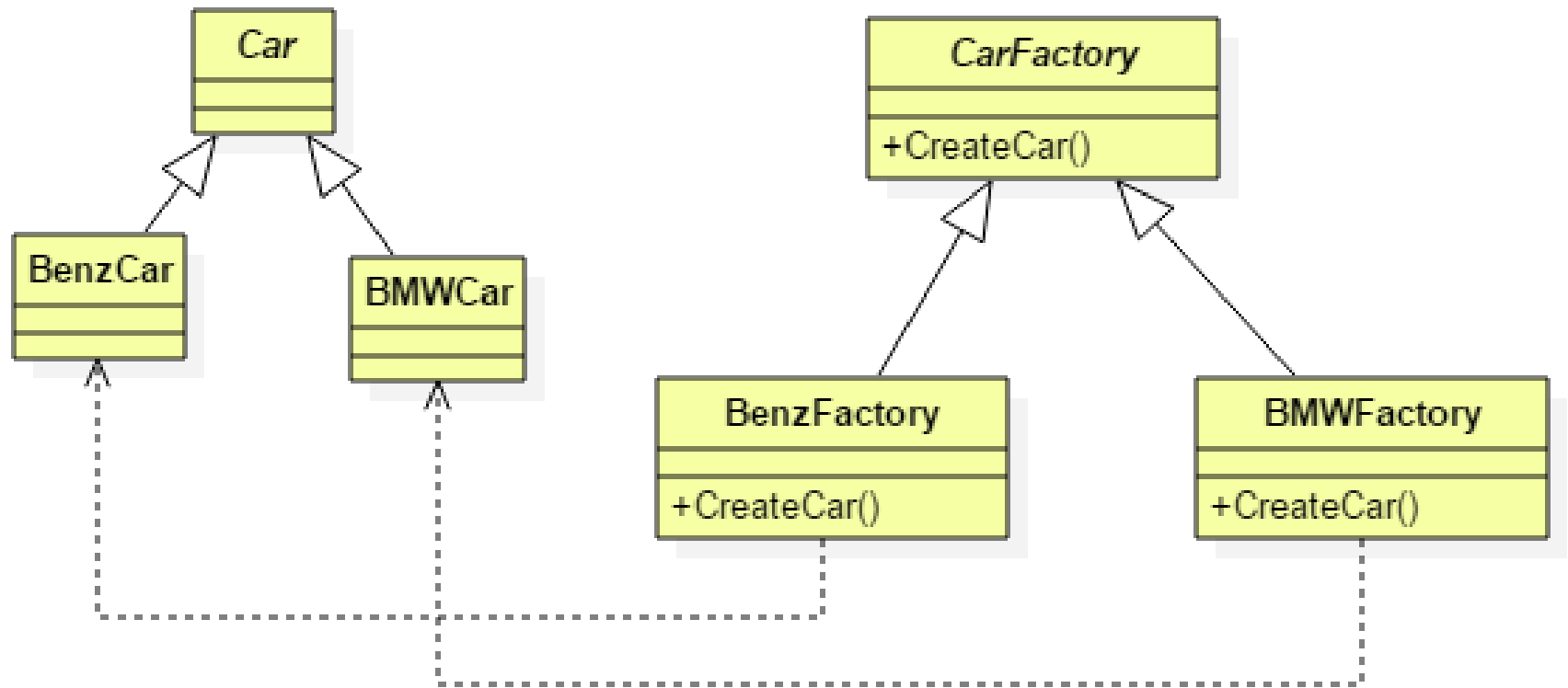


BMW



如果先不考虑不同的车型？

- 用 UML 类图来表示



工厂方法模式可以继续改进吗？

- 解决思路

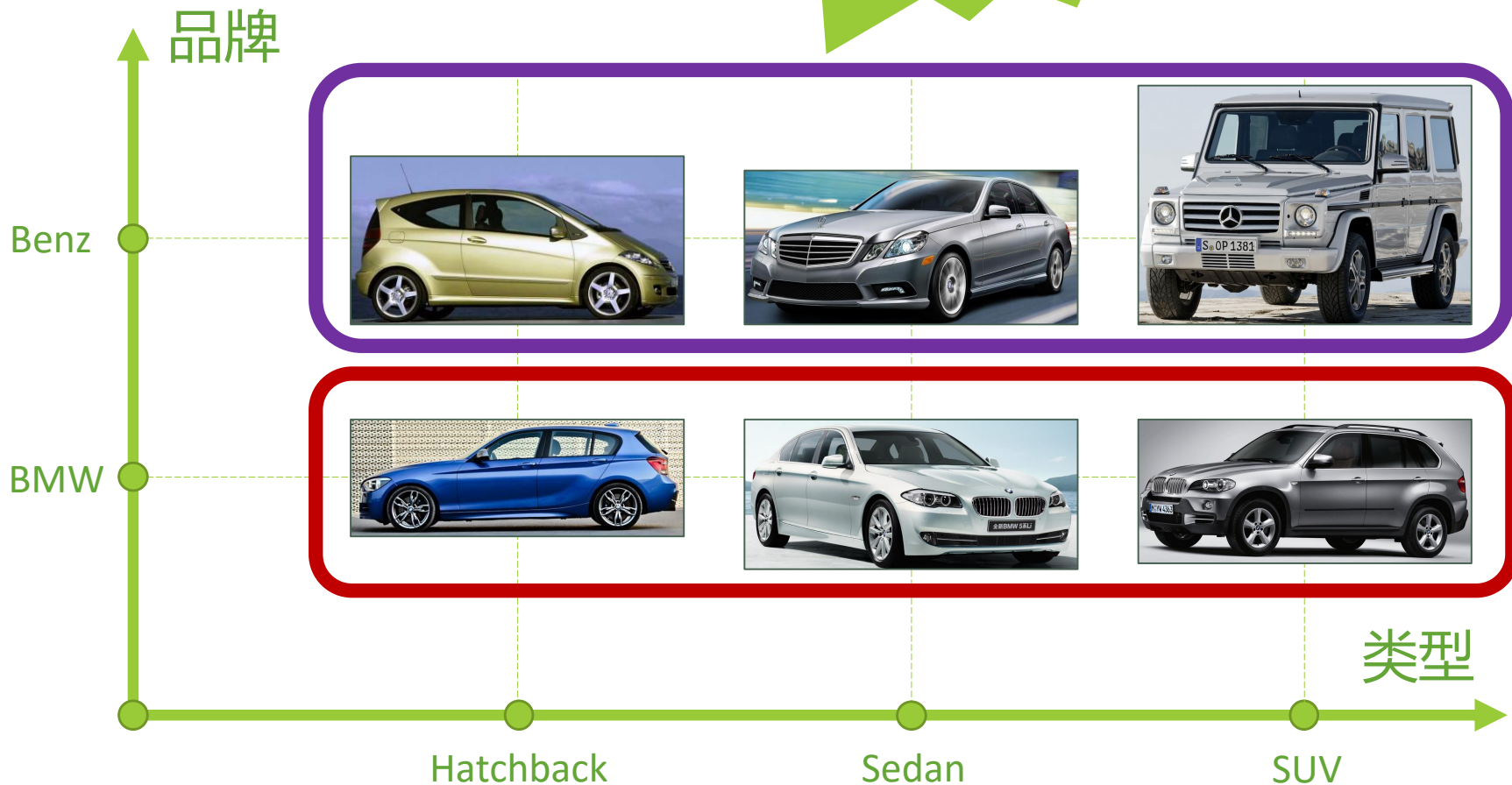
- 在工厂方法模式中，一个具体工厂只负责生产一种具体的产品，也就是说，**一个具体工厂中只有一个工厂方法**。如果我们希望提供一组产品，应该怎么做？

- **提供一组工厂方法**，每种工厂方法用于生产一种产品。这样，一个具体工厂就可以**生产一组产品**。

类似的问题

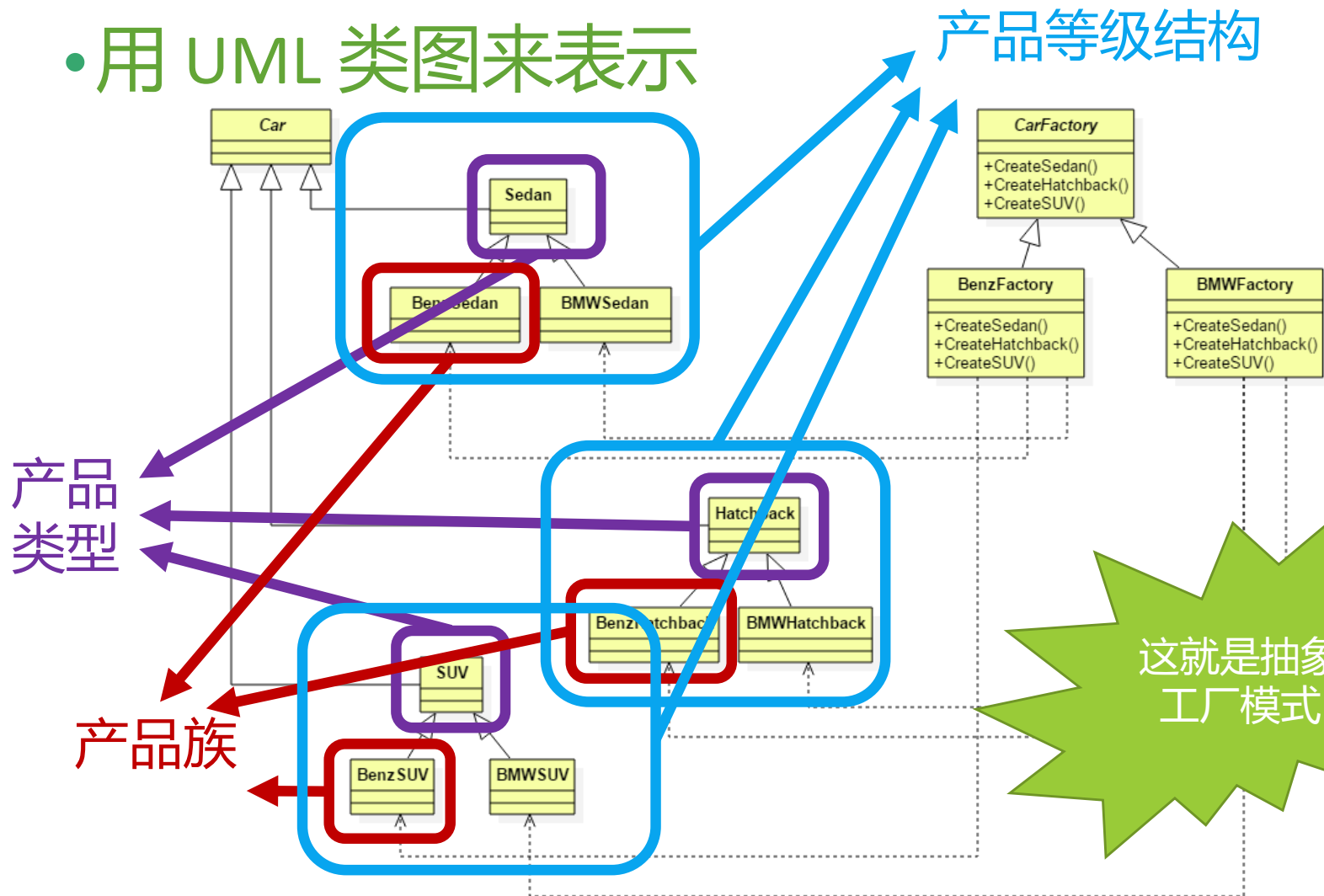
- 产品的维度

还有别的例子吗？



现在考虑不同的车型

- 用 UML 类图来表示



抽象工厂模式

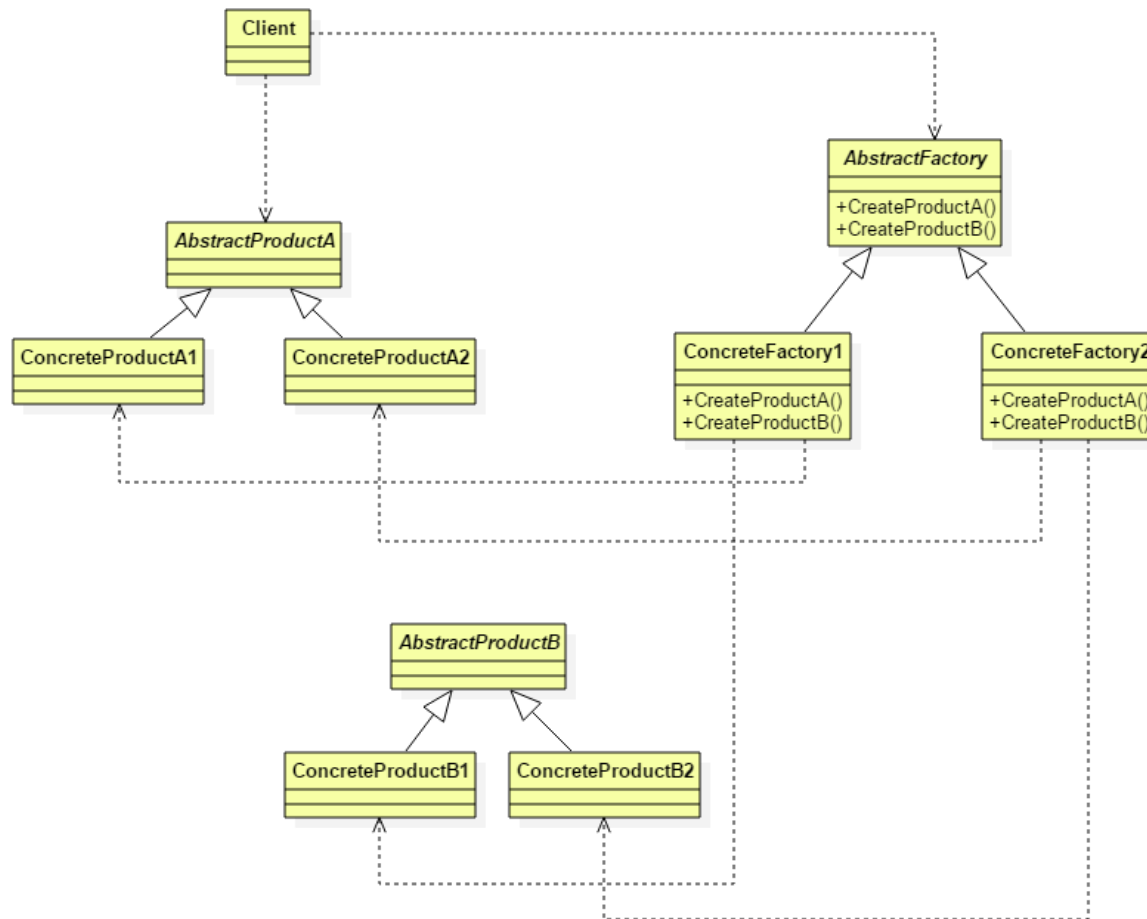
- 定义

- 提供了用于创建一系列相关的对象的接口，而无需定义它们的具体类。

- 也属于创建型模式，是各种工厂模式中最为抽象和通用的一种。

抽象工厂模式

- 抽象工厂模式 UML 类图

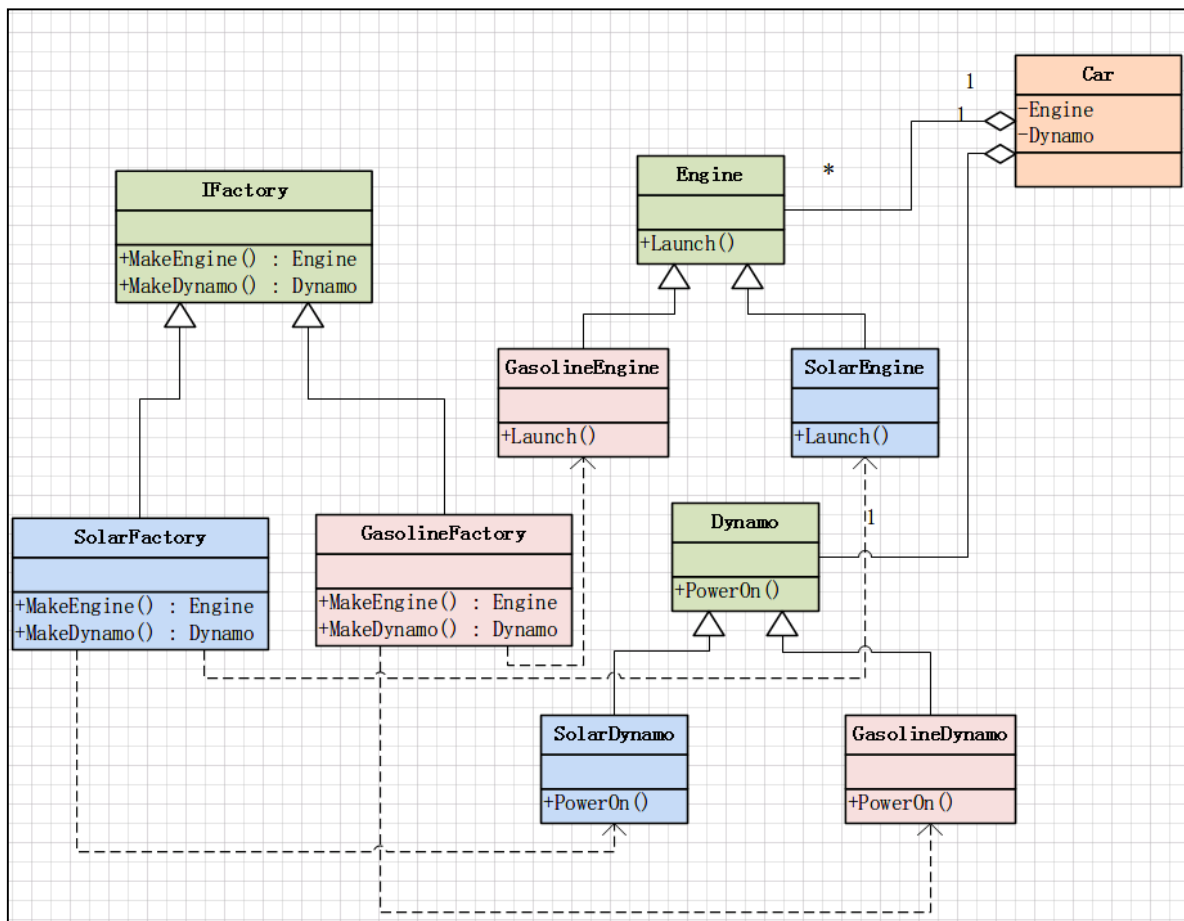


抽象工厂模式角色（与工厂方法类似）

- 抽象产品类（Abstract Product）
 - 定义了产品的共性，所有的具体产品都要实现这个接口。
- 具体产品类（Concrete Product）
 - 实现了产品接口的类，包含了业务逻辑，决定了产品在客户端中的具体行为。
- 抽象工厂类（Abstract Factory）
 - 也就是抽象工厂，是本模式的核心，定义了具体工厂类必须实现的接口。
- 具体工厂类（Concrete Creator）
 - 实现了如何具体创建产品类。有多少种产品，就要有多少个具体工厂。

引入抽象工厂模式的不同动力小汽车例子

- UML 类图



不同动力小汽车代码实现

- IFactory 接口 (抽象工厂)

```
interface IFactory
{
    Engine MakeEngine();
    Dynamo MakeDynamo();
}
```

不同动力小汽车代码实现

- GasolineFactory (具体工厂)

```
class GasolineFactory : IFactory
{
    public Engine MakeEngine()
    {
        return new GasolineEngine();
    }

    public Dynamo MakeDynamo()
    {
        return new GasolineDynamo();
    }
}
```

不同动力小汽车代码实现

- SolarFactory (具体工厂)

```
class SolarFactory : IFactory
{
    public Engine MakeEngine()
    {
        return new SolarEngine();
    }

    public Dynamo MakeDynamo()
    {
        return new SolarDynamo();
    }
}
```

不同动力小汽车代码实现

- Engine (抽象产品 A)

```
abstract class Engine
{
    public abstract void Launch();
}
```

不同动力小汽车代码实现

- GasolineEngine (具体产品 A1)

```
class GasolineEngine : Engine
{
    public override void Launch()
    {
        Console.WriteLine("Gasonline engine starts...");
    }
}
```

- SolarEngine (具体产品 A2)

```
class SolarEngine : Engine
{
    public override void Launch()
    {
        Console.WriteLine("Solar engine starts...");
    }
}
```


不同动力小汽车代码实现

- Dynamo (抽象产品 B)

```
abstract class Dynamo
{
    public abstract void PowerOn();
}
```

不同动力小汽车代码实现

- GasolineDynamo (具体产品 B1)

```
class GasolineDynamo : Dynamo
{
    public override void PowerOn()
    {
        Console.WriteLine("Gasoline electricity");
    }
}
```

- SolarDynamo (具体产品 B2)

```
class SolarDynamo : Dynamo
{
    public override void PowerOn()
    {
        Console.WriteLine("Solar electricity");
    }
}
```

不同动力小汽车代码实现

- Car

```
class Car
{
    private Engine _engine;
    private Dynamo _dynamo;

    public Engine Engine
    {
        get { return _engine; }
        set { _engine = value; }
    }

    public Dynamo Dynamo
    {
        get { return _dynamo; }
        set { _dynamo = value; }
    }

    public void Drive()
    {
        _dynamo.PowerOn();
        _engine.Launch();

        Console.WriteLine("Car is running");
    }
}
```

不同动力小汽车代码实现

- Client (具体产品 B1)

```
private static void Main(string[] args)
{
    //创建太阳能工厂对象，用于生产太阳能汽车所需的发动机与发电机
    IFactory factory = new SolarFactory();
    //如需生产汽油汽车所需的发动机与发电机，则将对象改为GasolineFactory
    //IFactory factory = new GasolineFactory();

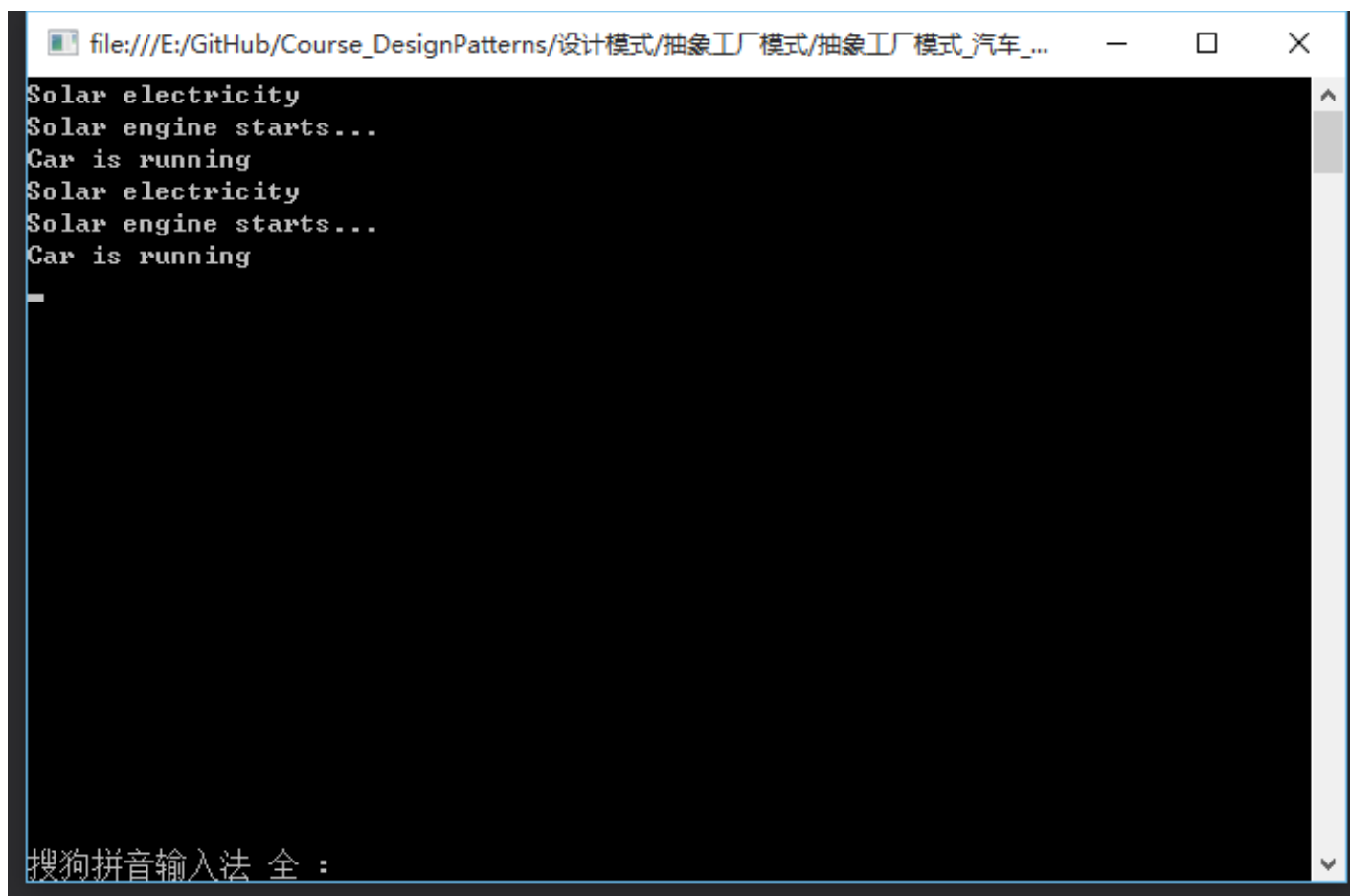
    Car car1 = new Car();
    car1.Engine = factory.MakeEngine();
    car1.Dynamo = factory.MakeDynamo();
    car1.Drive();

    Car car2 = new Car();
    car2.Engine = factory.MakeEngine();
    car2.Dynamo = factory.MakeDynamo();
    car2.Drive();

    Console.ReadLine();
}
```

不同动力小汽车代码实现

- 程序运行结果



A screenshot of a terminal window with a black background and white text. The window title bar shows the file path: file:///E:/GitHub/Course_DesignPatterns/设计模式/抽象工厂模式/抽象工厂模式_汽车_... . The output text is as follows:

```
Solar electricity  
Solar engine starts...  
Car is running  
Solar electricity  
Solar engine starts...  
Car is running  
_
```

At the bottom of the terminal, there is a text input field with the placeholder text "搜狗拼音输入法 全：".

抽象工厂模式在实际中的应用

- 背景介绍

- 某软件公司为一家企业做了一个电子商务网站，采用 **SQL Server** 作为数据库；现在，公司接到了另外一家企业的项目，需求类似，但是要求使用 **MySQL** 作为数据库，问应该如何进行改造？

- 不同数据库可能会带来的问题：

- 不同的数据库连接字符串（ MSSQL vs MySQL ）
- 不同的语法与数据类型（ 参考1， 参考2 ）

所以我们需要修改的代码将会有很多。

抽象工厂模式在实际中的应用

- 如何解决？

- 将数据本身以及对于数据的操作封装起来。这也是**分层式架构**的基础。

抽象工厂模式在实际中的应用

- 最基本的数据库访问程序
 - 要求采用 OOP 及分层的思想对数据库中的数据进行操作。
- 首先建立一个 User 类 (Model) , 用于存放数据库中 User 表中的数据。 (相当于用一个内存中的 User 对象对应数据库中 User 数据表中的一条记录)

```
class User
{
    public int ID { get; set; }
    public string Name { get; set; }
}
```


抽象工厂模式在实际中的应用

- IUser 接口

- 用于定义 User 对象可以进行的操作（即对 User 数据记录的操作）。

```
interface IUser
{
    void Insert(User user);
    User GetUser(int id);
}
```

抽象工厂模式在实际中的应用

- SqlserverUser 类
 - 用于在 SQL Server 数据库中操作 User 表 (省略了实际中使用的代码)

```
class SqlserverUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在SQL SERVER中给User表增加一条记录");
    }

    public User GetUser(int id)
    {
        User user = new User();
        Console.WriteLine("在SQL SERVER中根据ID得到User表一条记录");
        return user;
    }
}
```

抽象工厂模式在实际中的应用

- MysqlUser 类

- 用于在 MySQL 数据库中操作 User 表（省略了实际中使用的代码）

```
class MysqlUser : IUser
{
    public void Insert(User user)
    {
        Console.WriteLine("在MySQL中给User表增加一条记录");
    }

    public User GetUser(int id)
    {
        User user = new User();
        Console.WriteLine("在MySQL中根据ID得到User表一条记录");
        return user;
    }
}
```

抽象工厂模式在实际中的应用

- IFactory 接口

- 用于定义工厂类的接口，包含了所有的工厂方法。

```
interface IFactory
{
    IUser CreateUser();
}
```

抽象工厂模式在实际中的应用

- SqlserverFactory 类
 - 实现 IFactory 接口，用于实例化一个 SqlserverUser (DAL) 对象。

```
class SqlserverFactory : IFactory
{
    public IUser CreateUser()
    {
        return new SqlserverUser();
    }
}
```

抽象工厂模式在实际中的应用

- MySQLFactory 类

- 实现 IFactory 接口，用于实例化一个 MysqlUser 对象。

```
class MysqlFactory : IFactory
{
    public IUser CreateUser()
    {
        return new MysqlUser();
    }
}
```

抽象工厂模式在实际中的应用

- 客户端程序
 - 用于完成业务逻辑

如果要更换
数据库类型？

```
private static void Main(string[] args)
{
    User user = new User();

    //创建在SQL Server环境下使用的数据对象
    IFactory factory = new SqlserverFactory();

    IUser u = factory.CreateUser();
    u.Insert(user);
    u.GetUser(1);

    Console.ReadLine();
}
```

只需要修改此处代码

抽象工厂模式在实际中的应用

• 修改后的代码

```
private static void Main(string[] args)
{
    User user = new User();

    //创建在MySQL环境下使用的数据对象
    IFactory factory = new MysqlFactory();

    IUser u = factory.CreateUser();
    u.Insert(user);
    u.GetUser(1);

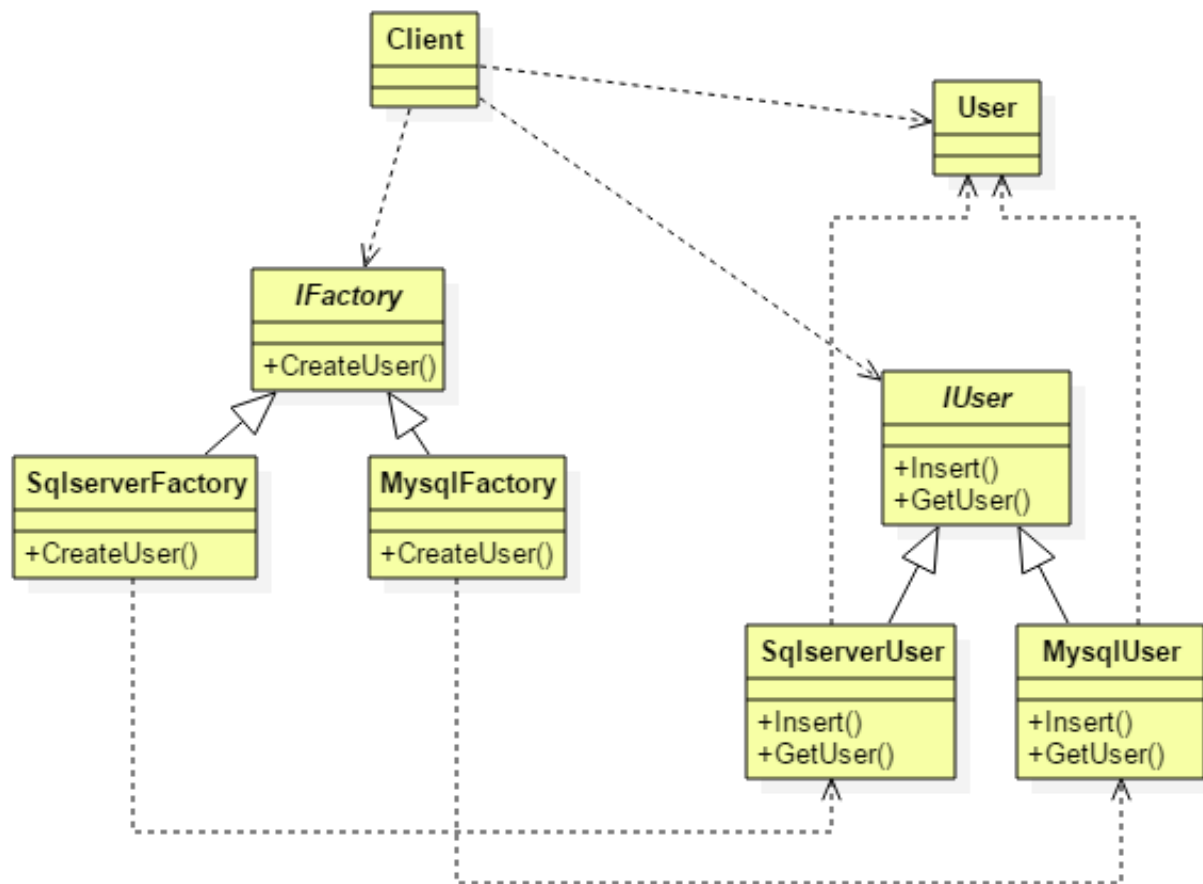
    Console.ReadLine();
}
```



如果数据库中不止有 User 一张数据表，如 Department 表？

抽象工厂模式在实际中的应用

- UML 类图



工厂模式在实际中的应用

- 需要增加的类有

1. Department 类 (Model)
2. IDepartment 接口 (IDAL)
3. SqlserverDepartment 类 (DAL)
4. MysqlDepartment 类 (DAL)

- 需要修改的类有

1. IFactory 类 (IFactory) : 增加 CreateDepartment() 方法 , 返回一个 IDepartment 对象。
2. SqlserverFactory 与 MysqlFactory 类 (Factory) : 实现 CreateDepartment() 方法。

增加了对Department表的支持后的代码

- IDepartment 接口

```
interface IDepartment
{
    void Insert(Department department);

    Department GetDepartment(int id);
}
```

增加了对Department表的支持后的代码

- SqlserverFactory 类

```
class SqlserverFactory : IFactory
{
    public IUser CreateUser()
    {
        return new SqlserverUser();
    }

    public IDepartment CreateDepartment()
    {
        return new SqlserverDepartment();
    }
}
```

增加了对Department表的支持后的代码

- MysqlFactory 类

```
class MysqlFactory : IFactory
{
    public IUser CreateUser()
    {
        return new MysqlUser();
    }

    public IDepartment CreateDepartment()
    {
        return new MysqlDepartment();
    }
}
```

增加了对Department表的支持后的代码

- SqlserverDepartment 类

```
class SqlserverDepartment : IDepartment
{
    public void Insert(Department department)
    {
        Console.WriteLine("在SQL SERVER中给Department表增加一条记录");
    }

    public Department GetDepartment(int id)
    {
        Department dept = new Department();
        Console.WriteLine("在SQL SERVER中根据ID得到Department表一条记录");
        return dept;
    }
}
```

增加了对Department表的支持后的代码

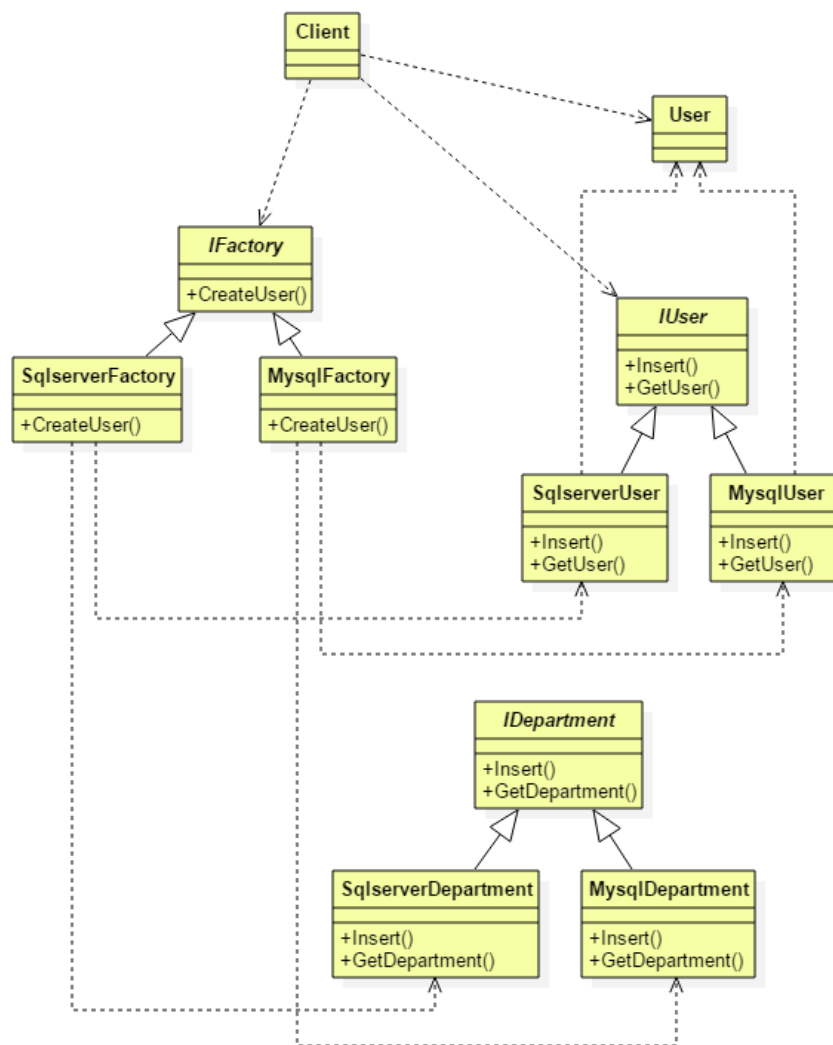
- MysqlDepartment 类

```
class MysqlDepartment : IDepartment
{
    public void Insert(Department department)
    {
        Console.WriteLine("在MySQL中给Department表增加一条记录");
    }

    public Department GetDepartment(int id)
    {
        Department dept = new Department();
        Console.WriteLine("在MySQL中根据ID得到Department表一条记录");
        return dept;
    }
}
```

增加了对Department表的支持后的代码

- UML 类图



抽象工厂模式总结

- 优点

- 具体产品从客户代码中分离。
- 容易改变产品的系列。
- 将一个系列的产品族统一到一起创建。

- 缺点

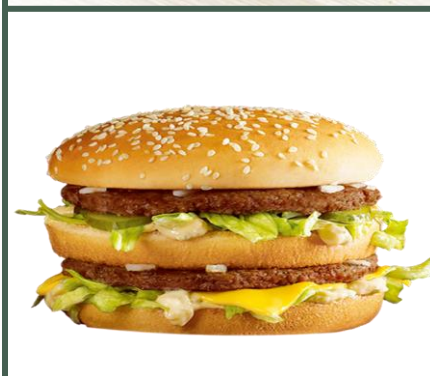
- 在产品族中**增加新的产品类型**较为困难，因为需要修改已有的抽象工厂类，在一定程度上**违反了“开放—封闭原则”**。

作业

- 用程序模拟以下产品线并代码实现



Burger



Chicken



Drink

现磨咖啡



作业

- 具体要求

- 画出程序的 UML 类图，推荐工具有：
 1. Microsoft Visio 2013/2016
 2. StarUML (<http://staruml.io/>) 。
 3. ProcessOn (<https://www.processon.com/>)
- 用任意一种 OOP 编程语言 (C++/Java/C#) 实现，要求必须能够运行。
- 以写**博客**的形式 ([范例1](#) , [范例2](#) , [范例3](#)) ，将**类图**及**所有源代码**，发布于博客园 (<http://www.cnblogs.com>) ，网址交由课代表 ([包存斌](#)) 汇总。
- 如果能够将代码同步提交到代码托管网站 **GitHub** (<http://www.github.com>) ，**加分！** (必须提供 **GitHub** 发布代码的网址，方法自学 ([参考](#) ，也可使用 **GitExtensions** 等 GUI 辅助工具) ，**仅上传压缩包无效**)
- 截止时间：2015年11月9日