一、单例模式

1.1 设计模式代码

（1）

```
import datetime
import Goods
import Customers


class Cart:
    totalPrice = 0
    createdTime = ''
    good = []
    customer = Customers.Customer()
    totalNum = 0


    def __init__(self, place, time, num):
        self.createdTime = datetime.datetime.now().isoformat()
        self.totalNum = len(self.good)
        self.totalPrice = self.GetPrice()

    def GetPrice(self):
        total = 0
        for i in self.good:
            total += i.price
        return total

    def GetNum(self):
        return len(self.good)

    def AddGood(self, theGood):
        self.good.append(theGood)


cart = Cart()
```

在 Cart.py 中创建 Cart 类的一个对象

（2）

```
import sys
sys.path.append('..')
from Entity.Goods import Good
from Entity.Cart import cart
```

在其他文件中通过

```
from Entity.Cart import cart
```

导入 cart 变量，从而这个 cart 变量就是单例变量

1.2 设计思路分析

其实，Python 的模块即使天然的单例模式。这是因为模块在第一次被导入时，会生成.pyc 文件，而在之后的导入过程中会直接加载.pyc 文件而不会在此执行模块代码。因此，只需将相关的函数和数据定义在一个模块中就可以获得一个单例对象。

在这个简单的电商系统中，购物车始终都只有一个，不管里面有没有商品或者商品有多少。因此我们将购物车设置为单例变量。

1.3 优势

单例模式（Singleton Pattern）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。当我们希望在整个系统中，某个类只能出现一个实例时，单例对象就能排上用场。

如果在系统中很多地方都需要使用某一个类的对象，那么在很多地方都需要创建这种对象的实例，从而在系统中存在多个该类的实例对象，从而严重浪费内存资源。而实际上我们只需要通过一个实例对象作为单例对象就可以达到预期的效果。从而大量节省内存资源。同时，单例模式还可以避免对资源的多重占用。

二、外观模式

2.1 设计模式代码

```
# 外观模式
class OpCart:

    '''
    示例:
    opcart = OpCart()
    opcart.op(car, 1)  # 添加一个car商品
    opcart.op(car, -1)  # 删除一个car商品
    '''

    def __init__(self, cart):
        self.cart = cart

    # 判断是否存在该商品
    def __exist(self, good):
        for goods in self.cart.good:
            if goods.good == good:
                return True
        return False

    # 返回商品的索引
    def __return_index(self, index, good):
        for goods in self.cart.good:
            if goods.good == good:
                return index
            index += 1

    # 添加或删除操作
    def op(self, good, num):
        mydb = MysqldbHelper()
        table = 'cart'+cart.customer.customerId
```

```
        if self.__exist(good):  # 如果商品已经在购物车中，进行数
量的修改
            index = self.__return_index(0, good)
            if num + self.cart.good[index].num <= 0:  # 如果最
后该商品的个数小于等于 0，则删除该商品
                del self.cart.good[index]
                params = {'name':
cart.good[index].good.goodName, 'id':
cart.good[index].good.goodId}
                mydb.delete(table, params)
            else:
                self.cart.good[index].num += num  # 如果最后该商
品的个数大于 0，则修改该商品个数
                params = {'name': cart.good[index].good.goodName,
'id': cart.good[index].good.goodId, 'num':
cart.good[index].num}
                cond_dict = {'name':
cart.good[index].good.goodName, 'id':
cart.good[index].good.goodId, 'num': cart.good[index].num+num}
                mydb.update(table, cond_dict, params)
        else:
            if num > 0:  # 如果购物车不存在该商品且 num 大于 0，则
放入购物车
                goods = Goods(good, num)
                cart.AddGood(goods)
                params = {'name': good.goodName, 'id':
good.goodId, 'num': num}
                mydb.insert(table, params)
            else:
                print("购物车中不存在该商品，无法删减")
```

2.2 设计思路分析

购物车是项目的重要内容，对购物车进行增删改的操作时，需要设计对购物车的内容修改，对数据库进行操作等。为了简化对购物车的操作流程，我们使用了外观模式，将对购物车的操作整合到了一起，即 OpCart 类，具体操作时，只需输入商品对象以及增加或减少的个数即可。

2.3 优势

外观模式是一种使用频率非常高的结构型设计模式，它通过引入一个外观角色来简化客户端与子系统之间的交互，为复杂的子系统调用提供一个统一的入口，降低子系统与客户端的耦合度，且客户端调用非常方便。我们这里通过引入一个 OpCart 类来整合对购物车的操作，简化了流程，秩序调用一个方法即可实现对购物车的增删以及对数据库的增删等操作。

三、组合模式

3.1 设计模式代码

（1）

```
# 组合模式
class Goods:
    def __init__(self, good, num):
        self.good = good
        self.num = num
```

```
    def get_price(self):
        return num * good.goodPrice
```

3.2 设计思路分析

在 Good 商品类的基础上，再设计一个 Goods 类，将不同的商品对象整合为一个对象，便于之后存储在购物车中并进行增删操作。将不同的商品看作相同的 Goods 类的对象之后，在购物车中的操作，例如增加、删除商品，增加、减少商品数量的操作都可以统一起来。

3.3 优势

组合模式（Composite Pattern），又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。这里将我们的各式各样的商品类的对象整合到一个 Goods 类的对象。具体的，例如三件相同尺寸的衣服选择加入购物车时，如果将三个衣服对象放入购物车，不仅麻烦，而且后续的增删也很复杂，因此，组合模式方便对整个层次结构进行控制，可以很大程度上简化购物车的相关操作。

四、四、工厂模式

4.1 设计模式代码

（1）

```
# 工厂模式
class AbsFactory:
    @abstractmethod
    def produce(self, name:str):
        pass


class FoodFactory(AbsFactory):
    def produce(self, name:str):
```

```
        if name == 'Snack':
            return Goods.Snack()
        if name == 'Drink':
            return Goods.Drink()


class ClotherFactory(AbsFactory):
    def produce(self, name:str):
        if name == 'Shirt':
            return Goods.Shirt()


class ElectricityFactory(AbsFactory):
    def produce(self, name:str):
        if name == 'Phone':
            return Goods.Phone()
        if name == 'Computer':
            return Goods.Computer()
```

4.2 设计思路分析

根据依赖倒置原则，将不同的商品抽象为食品、衣物、电子产品等，从底端开始抽象，让上层模块不依赖底层模块。在这里创建多个工厂分别生产不同种类的产品，而在类中并不关心这些产品的具体原料等，所以采用工厂模式让对象的创建和使用分离。这个工厂实际上是抽象工厂，允许平台通过抽象接口获得一组相关产品。

4.3 优势

采用抽象工厂模式分离了具体的类，客户端通过抽象接口操纵实例。类名也在具体工厂的实现中被分离，客户端不必调用。这样也有利于多个产品的批量生成。具体而言，通过抽象工厂定义 produce 的函数，在具体工厂中重写这一函数，以产生不同的对象供客户端调用，这样客户端就不用知道类名和类的具体参数就能够得到对应的“产品”。