基础部分

1. 设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。（设计模式、设计经验的总结、为了重用代码）
2. 面向对象设计原则：（设计原则、对接口编程、优先使用对象）

- 对接口编程而不是对实现编程。
- 优先使用对象组合而不是继承。

3. 设计模式的六大原则　　（六大原则）

(1)　对扩展开放，对修改关闭：在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。（对扩展开放、对修改关闭）

(2)　里氏代换原则：任何基类可以出现的地方，子类一定可以出现。里氏代换原则是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。（里氏代换、子类一定可以出现）

(3)　依赖倒转原则：针对接口编程，依赖于抽象而不依赖于具体。（依赖倒转、对接口编程）

(4)　接口隔离原则：使用多个隔离的接口，比使用单个接口要好。（接口隔离）

(5)　迪米特法则，又称最少知道原则：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。（最少知道原则、相对独立）

(6)　合成复用原则：尽量使用合成/聚合的方式，而不是使用继承。（合成复用、使用合成方式）

4. 设计模式大概可以分为三大类：创建型模式、结构型模式、行为型模式。（三大类、创建、结构、行为）

第 1 部分 创建模式

1. 在工厂设计模式中，不同的输入返回不同的类，也就是说，不同的输入选择不同的子类来生成这个对象。（工厂模式、不同输入、返回不同类）
2. 工厂通常有两种形式：一种是工厂方法，它是一个方法（即函数），对不同的输入参数返回不同的对象；第二种是抽象工厂，它是一组用于创建一系列相关事物对象的工厂方法。（两种方式、工厂方法、抽象工厂）

(1) 工厂方法简单举例：

```
class JSONConnector:
    def __init__(self, filepath):

class XMLConnector:
    def __init__(self, filepath):
```

```
def connector_factory(filepath):
    if filepath.endswith('json'):
        connector = JSONConnector
    elif filepath.endswith('xml'):
        connector = XMLConnector
    else:
        raise ValueError('Cannot connect to {}'.format(filepath))
    return connector(filepath)

factory = None
try:
    factory = connection_factory(filepath)
except ValueError as ve:
    print(ve)
return factory
```

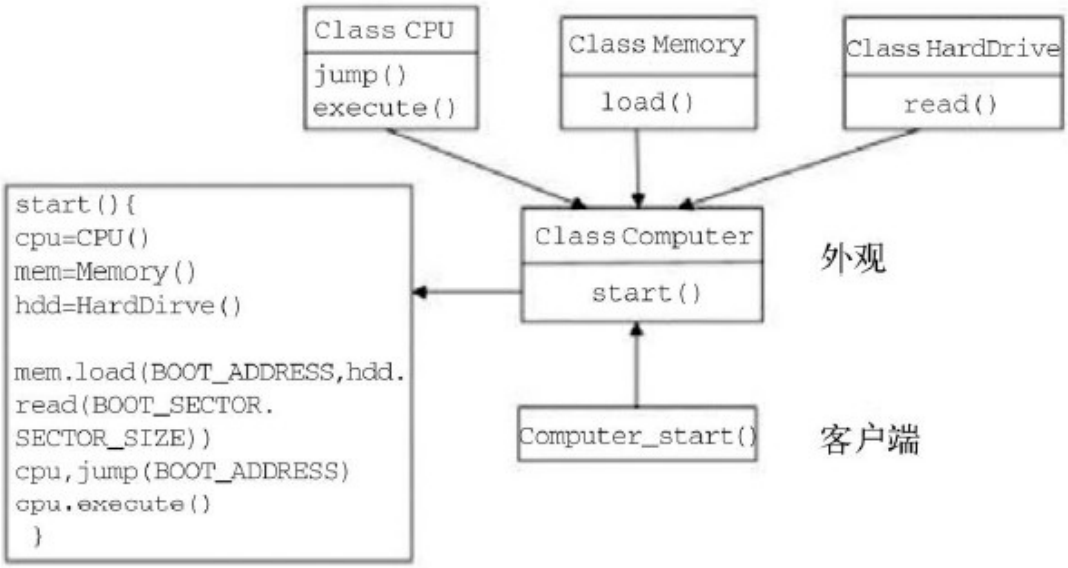
connector_factory 是一个工厂方法。

(2) 抽象工厂设计模式是抽象方法的一种泛化。概括来说，一个抽象工厂是一组工厂方法，其中的每个工厂方法负责产生不同种类的对象。（抽象工厂、一组工厂方法）

3. 建造者设计模式：即一个对象由多个部分组成，只有当各个部分都创建好，这个对象才算是完整的设计模式。类似 html 页面中的 head 和 body 部分，只有 head 和 body 部分健全，才是一个完整的页面。（建造者、多个部分组成）
4. 建造者模式有两个参与者：建造者和指挥者。通俗地说，建造者指整个类，而指挥者指类下面用于填充数据的函数。（建造者、类、指挥者、函数）
5. 新电脑类比的例子有助于区分建造者模式和工厂模式：购买现成电脑就是工厂模式，组装电脑是建造者模式。（现成电脑、工厂模式、组装电脑、建造者模式）
6. 原型设计模式帮助我们创建对象的克隆，其最简单的形式就是一个 clone()函数，接受一个对象作为输入参数，返回输入对象的一个副本。在 Python 中，这可以使用 copy.deepcopy()函数来完成。（原型模式、克隆、副本、deepcopy()）

第 2 部分 结构模式

1. 适配器模式是一种结构型设计模式，帮助我们实现两个不兼容接口之间的兼容。（适配器模式、兼容接口）
2. 装饰器模式能够以透明的方式（不会影响其他对象）动态地将功能添加到一个对象中。（装饰器模式、动态地添加对象）
3. 外观设计模式有助于隐藏系统的内部复杂性，并通过一个简化的接口向客户端暴露必要的部分。本质上，外观（Facade）是在已有复杂系统之上实现的一个抽象层。（外观模式、隐藏内部复杂性）



从图中展示的类可知，仅 Computer 类需要暴露给客户端代码。客户端仅执行 Computer 的 start()方法。所有其他复杂部件都由外观类 Computer 来维护。

4. 也就是说，外观模式事实上就是将统一接口。（外观模式、统一接口）

5. 享元设计模式通过为相似对象引入数据共享来最小化内存使用，提升性能。一个享元就是一个包含状态独立的不可变（又称固有的）数据的共享对象。例如，在《反恐精英》游戏中，同一团队（反恐精英或恐怖分子）的所有士兵看起来都是一样的（外在表现）。同一个游戏中，（两个团队的）所有士兵都有一些共同的动作，比如，跳起、低头等（行为）。这意味着我们可以创建一个享元来包含所有共同的数据。（享元设计模式、共享对象、共享的动作）

6. 若想要享元模式有效，需要满足以下几个条件。（几个条件、大量对象、存储代价太大、对象 ID 不重要）

(1) 应用需要使用大量的对象。

(2) 对象太多，存储/渲染它们的代价太大。一旦移除对象中的可变状态（因为在需要之时，应该由客户端代码显式地传递给享元），多组不同的对象可被相对更少的共享对象所替代。

(3) 对象 ID 对于应用不重要。对象共享会造成 ID 比较的失败，所以不能依赖对象 ID。

7. MVC 被认为是一种架构模式而不是一种设计模式。架构模式与设计模式之间的区别在于前者比后者的范畴更广。

- 控制器：负责转发请求，对请求进行处理。（控制器、处理请求）
- 视图：界面设计人员进行图形界面设计。（视图、图形界面设计）
- 模型：程序员编写程序应有的功能（实现算法等等）、数据库专家进行数据管理和数据库设计(可以实现具体的功能)。（模型、实现功能）

8. MVC 模式提供了以下这些好处。（好处）

(1) 视图与模型的分离允许美工一心搞 UI 部分，程序员一心搞开发，不会相互干扰。（不会相互干扰）

(2) 由于视图与模型之间的松耦合，每个部分可以单独修改/扩展，不会相互影响。

例如，添加一个新视图的成本很小，只要为其实现一个控制器就可以了。（扩展成本小、只添加控制器）

(3) 因为职责明晰，维护每个部分也更简单。（维护简单）

9. 代理设计模式有四种：（代理设计有四种）

(1) 远程代理：实际存在于不同地址空间（例如，某个网络服务器）的对象在本地的代理者。（远程代理、表面本地、实际上是远程对象）

(2) 虚拟代理：用于懒初始化，将一个大计算量对象的创建延迟到真正需要的时候进行。（虚拟代理、延迟创建）

(3) 保护/防护代理：控制对敏感对象的访问。（防护代理、敏感对象）

(4) 智能代理：在对象被访问时执行额外的动作。此类代理的例子包括引用计数和线程安全检查。（智能代理、执行额外的动作）

第 3 部分 行为型模式

1. 责任链模式用于让多个对象来处理单个请求时，或者用于预先不知道应该由哪个对象（来自某个对象链）来处理某个特定请求时。（责任链模式、对象链）

目的：避免请求发送者与接收者耦合在一起，让多个对象都可接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。（可接收请求的对象、连接成对象链）

其原则如下所示。

(1) 存在一个对象链（链表、树或任何其他便捷的数据结构）。

(2) 我们一开始将请求发送给链中的第一个对象。

(3) 对象决定其是否要处理该请求。

(4) 对象将请求转发给下一个对象。

(5) 重复该过程，直到到达链尾。

2. 命令设计模式帮助我们将一个操作（撤销、重做、复制、粘贴等）封装成一个对象。简而言之，这意味着创建一个类，包含实现该操作所需要的所有逻辑和方法。（命令模式、操作封装成对象）

3. DSL 指一种领域特定语言，是一种针对一个特定领域的有限表达能力的计算机语言。很多不同的事情都使用 DSL，比如，战斗模拟、记账、可视化、配置、通信协议等。DSL 分为内部 DSL 和外部 DSL。（DSL、领域特定语言）

(1) 内部 DSL 构建在一种宿主编程语言之上。（内部、依赖宿主语言）

(2) 外部 DSL 不依赖某种宿主语言。例如，在 python 中，mysql 语言就是一种外部 DSL。

4. 解释器模式仅与内部 DSL 相关。因此，我们的目标是使用宿主语言提供的特性构建一种简单但有用的语言，在这里，宿主语言是 Python。也就是说，解释器模式就是在 python 中创建一种内部简单语言来简化代码，而使用 python 来解释这种语言。（解释器模式、创建简单语言、python 解释语言）

5. 观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。（观察者模式、一对多关系、状态改变、自动更新）
6. 状态模式：类的行为是基于它的状态改变的。也就是说，允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。（状态模式、类的行为、基于状态而改变）
7. 策略模式：一个类的行为或其算法可以在运行时更改。定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。（策略模式、封装算法、可相互替换）
8. 模板模式：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。（模板模式、定义骨架、在子类实现步骤）