- 写出OOA、OOD、OOP的涵义（20字左右）

  > - OOA: 面向对象分析
  > - OOD: 面向对象设计
  > - OOP: 面向对象编程

- 写出七大设计原则

  > 1. 单一职责原则(一个类只负责一个功能领域中的相应职责)
  > 2. 开闭原则(对扩展开放，对修改关闭)
  > 3. 里氏替换原则(所有引用基类对象的地方能够透明地使用其子类的对象)
  > 4. 依赖倒转原则(抽象不应该依赖于细节，细节应该依赖于抽象)
  > 5. 接口隔离原则(使用多个专门的接口，而不使用单一的总接口)
  > 6. 合成利用原则 (尽量使用对象组合，而不是继承来达到利用的目的)
  > 7. 迪米特法则(一个软件实体应当尽可能少地与其他实体发生相互作用)

- 写出三类23种设计模式中至少10种设计模式的名称和类别

  > - 创建型
  >
  >   1. 简单工厂模式
  >   2. 抽象工厂模式
  >   3. 单例模式
  >   4. 构建者模式
  >   5. 原型模式
  >
  > - 结构型
  >
  >   1. 适配器模式
  >   2. 装饰模式
  >   3. 代理模式
  >   4. 外观模式
  >   5. 桥接模式
  >   6. 组合模式
  >   7. 享元模式
  >
  > - 行为型
  >
  >   1. 模板方法模式
  >   2. 策略模式
  >   3. 观察者模式
  >   4. 中介者模式
  >   5. 状态模式
  >   6. 责任链模式
  >   7. 命令模式
  >   8. 迭代器模式
  >   9. 访问者模式
  >   10. 解释器模式
  >   11. 备忘录模式

- 写出单例模式中的三种保证线程安全的写法

```
public class Student01 {
 private static volatile Student01 student = null;
 private Student01() {
 }

 public static synchronized Student01 getInstance() {
     if (student == null) {
         student = new Student01();
     }
     return student;
 }
}
```

```
public class Student02 {
 private static volatile Student02 student = null;
 private Student02() {

 }

 public static Student02 getInstance() {
     if (student == null) {
         synchronized (Student02.class) {
             if (student == null) {
                 student = new Student02();
             }
         }
     }

     return student;
 }
}
```

```
public class Student03 {
 private Student03() {

 }

 private static class InnerStudent {
     static Student03 INSTANCE = new Student03();
 }

 public static Student03 getInstance() {
     return InnerStudent.INSTANCE;
 }
}
```

```
public class Student01 {
    private static volatile Student01 student = null;
    private Student01() {
    }

    public static synchronized Student01 getInstance() {
        if (student == null) {
            student = new Student01();
        }
        return student;
    }
}
```

```
public class Student02 {
    private static volatile Student02 student = null;
    private Student02() {

    }

    public static Student02 getInstance() {
        if (student == null) {
            synchronized (Student02.class) {
                if (student == null) {
                    student = new Student02();
                }
            }
        }

        return student;
    }
}
```

- 写出两种攻击单例模式的方式的名称

> 1. 反射代理;
> 2. 字节码序列化(只能针对实现了 Serialiable 接口的单例有效)

- 写出volatile可以解决的两种问题

> 1. 禁止重排序
>
> > 编译器对代码进行编译的时候会检查代码之间相互依赖问题，如果相互之间没有依赖关系，那么他们之间的执行顺序可能会被编译器优化掉。本来是要先执行A 指令再执行B 指令的。但是由于B 指令执行的时间更短更快，编译器就有可能 会进行优化让B 指令优先于A 指令执行。
>
> 2. 多核CPU 之间的可见性问题
>
> > 每一个CPU 中都有自己的高速缓存(寄存器缓存)，CPU 之间的这种缓存是相互独立不可见的。
> >
> > CPU 处理内存变量的工作分为：1. 加载内存数据到CPU 缓存; 2. 处理变量; 3. 写回数据到CPU。

> 这里如果同一个内存被多个CPU 同一时刻读取到调整缓存，那么他们都对这个变量进行了操作的话，这些CPU 相互之间都不知道其他CPU 对其做了什么操作，最后就可能影响到该内存数据的最终结果不是我们期望的结果。